SOFTWARE OBFUSCATION WITH SYMMETRIC CRYPTOGRAPHY

THESIS

Alan C. Lin, First Lieutenant, USAF

AFIT/GCS/ENG/08-15

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

AFIT/GCS/ENG/08-15

SOFTWARE OBFUSCATION WITH SYMMETRIC CRYPTOGRAPHY

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science (Computer Science)

Alan C. Lin, BS

First Lieutenant, USAF

March 2008

AFIT/GCS/ENG/08-15

SOFTWARE OBFUSCATION WITH SYMMETRIC CRYPTOGRAPHY

Alan C. Lin, BS

First Lieutenant, USAF

Approved:

_________/signed/____________________ ________
J. Todd McDonald, Lt Col, USAF (Chairman) Date

_________/signed/____________________ ________
Stuart Kurkowski, Lt Col, USAF (Member) Date

_________/signed/____________________ ________
Dr. Richard Raines (Member) Date

AFIT/GCS/ENG/08-15

Abstract

Software protection is of great interest to commercial industry. Millions of dollars and years of research are invested in the development of proprietary algorithms used in software programs. A reverse engineer that successfully reverses another company's proprietary algorithms can develop a competing product to market in less time and with less money. The threat is even greater in military applications where adversarial reversers can use reverse engineering on unprotected military software to compromise capabilities on the field or develop their own capabilities with significantly less resources. Thus, it is vital to protect software, especially the software's sensitive internal algorithms, from adversarial analysis.

Software protection through obfuscation is a relatively new research initiative. The mathematical and security community have yet to agree upon a model to describe the problem let alone the metrics used to evaluate the practical solutions proposed by computer scientists. We propose evaluating solutions to obfuscation under the intent protection model, a combination of white-box and black-box protection to reflect how reverse engineers analyze programs using a combination white-box and black-box attacks. In addition, we explore use of experimental methods and metrics in analogous and more mature fields of study such as hardware circuits and cryptography. Finally, we implement a solution under the intent protection model that demonstrates application of the methods and evaluation using the metrics adapted from the aforementioned fields of study to reflect the unique challenges in a software-only software protection technique.

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Lt Col J. Todd McDonald, for his guidance and support throughout the course of this thesis effort. The insight and experience was certainly appreciated. I would also like to thank AFRL/RYT and AFOSR for sponsoring our Program Encryption Group (PEG). I would also like to acknowledge fellow PEG members, Maj Ken Norman and Capt Moses James, for being a sounding board and a sanity check for my work.

Most importantly, I want to thank my loving, understanding, and patient wife for her support throughout these past months at AFIT.

Alan C. Lin

Table of Contents

Page

List of Figures

Page

List of Tables

Page

SOFTWARE OBFUSCATION WITH SYMMETRIC ENCRYPTION

I. Introduction

Information assurance is traditionally comprised of two components: network security and operation system integrity. Software protection is intended to complement this traditional viewpoint with an application-centric approach to protect Department of Defense (DoD) software critical to national security. The goal of this research is to improve protection of DoD scientific, engineering, and simulation software within their normal operating environment while minimizing impact on code performance and usability (Hughes and Stytz, 2003). This research attempts to provide a model and a technique for software protection that produces quantifiable protection against reverse-engineering analysis. The model and technique address unique difficulties in practical implementation of software application protection with a software-only approach. Methodology for this research adapts practices in functionally related fields such as data cryptography and hardware circuit protection.

The purpose of this chapter is to outline the efforts of this software protection research and will address the background, the investigated problem and the methodology. This paper intends to support AFRL's Software Protection Initiative (AFRL/SPI) in producing techniques that can measurably secure applications through integration with critical software.

1.1 Background

Reverse engineering is the process by which a fabricated product is deconstructed to understand its design, architecture, and underlying technological properties (Eilam, 2005:xxiv). Historically, this process was successful in assimilating the technological edge of an adversary. Famous and successful WWII examples include: American and British Jerry cans from German gasoline cans, Soviet R-7 rockets from German V2 rockets, and Soviet Tupolev Tu-4 strategic bomber from American B-29s as seen in Figure 1 (“Reverse,” 2007; “B-29,” 2007; “Tupolev,” 2007).

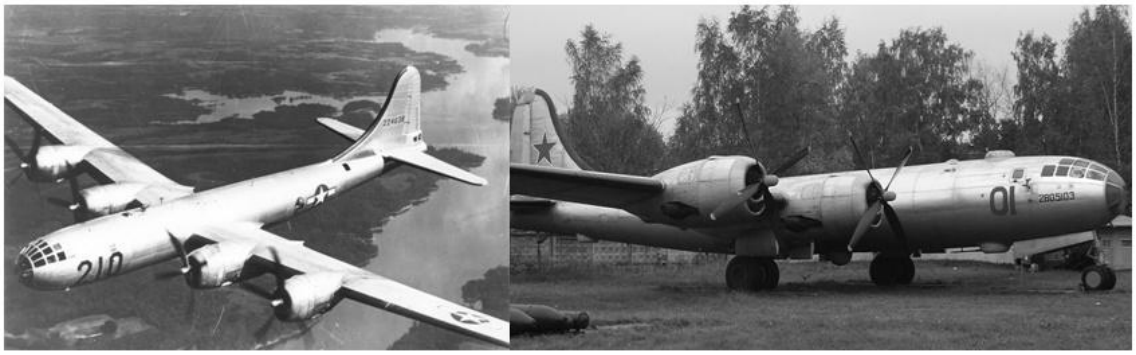


Figure 1. American B-29 (left) and Soviet Tu-4 (right)

In post-WWII events, the crash of an F-117A in 1999 during the Kosovo conflict resulted in the capture of a second-generation stealth platform by the Yugoslavians, who subsequently sold parts of the wreckage to the Russians (Richelson, 2001:62). While no official reports indicate that the Russians have made significant use of the wreckage, it is conceivable that the exploitation of F-117A technologies from the wreckage could be used to advance anti-stealth technologies, or sold to nation-states aspiring to advance their own stealth programs (Carlson, 1999). The crash of an US Navy EP-3 intelligence aircraft off the coast of China in 2001 is another often cited example of where military hardware was lost to foreign nation-states. In this instance, losing the hardware asset was potentially less severe than losing the software installed on the platform; CNN reports that the airplane's most sensitive components were the "software and encryption devices used for unscrambling military codes ("Plane," 2001)."

The old paradigm of software security is relatively weak. Previously, military technology software required computational power available only on high performance computing (HPC) assets that were safeguarded by export control laws. HPC trade regulations, therefore, indirectly protected all of the state-of-the-art defense applications. However, with the advancement of computer hardware technology, theft of software no longer necessitates theft of corresponding hardware (Hughes and Stytz, 2007). As an example, Xiaodong Meng, in 2006, pleaded guilty under espionage charges for exporting Quantum3D proprietary software products to China, Thailand, and Malaysia (Department of Justice, 2006); the international-trade restricted software products are exclusively used to train US military fighter pilots ("U.S. Arms Software," 2007).

As shown by the Meng incident, traditional information assurance protects the data stored on the system and the data travelling on the network, but not the high-valued applications that actually *generate* the data.

Many research areas credit software for achieving distinct technological advancements; investments made by the DoD over the last three decades have yielded technological advantages in electromagnetic modeling for radar signature predictions, fluid dynamic simulations for aircraft testing, and many other critical fields. The lack of quantifiable software protection mechanisms irresponsibly risks unauthorized exploitation of high value applications and threaten to erode our technological advantage. Without safeguards, adversaries can reverse our applications to develop countermeasures to our weapons, save on research and development costs, and build more advanced weapons (Hughes and Stytz, 2007).

Thus, in congruence with AFRL/SPI's mission statement, software technologies critical to national defense must be protected from reverse-engineering (Hughes and Stytz, 2007). The overall goal of this research is to develop an algorithm to protect software from reverse-engineering attacks and select metrics and benchmarks to quantifiably evaluate the developed algorithms. We examine the general software protection problem, analyze current interpretations and models of the problem, and study practical implementations of specific techniques. It is the intent of this research to describe a model that accurately reflects the software protection problem and demonstrate the quantifiable security of the proposed protection algorithm under the model using benchmark programs.

1.2 Problem Under Investigation

AFRL/SPI identifies four main thrust areas for research in long-term application security: algorithms, environments, benchmarks and metrics, and integration. Algorithms research addresses the need for techniques that produce protected software. Software environment research focuses on methods to protect software throughout its entire development process. Benchmarks and metrics are necessary so for consistent and accurate measurement of the potency and cost of proposed protection techniques. Integration research focuses on efficient composition of both multiple application security techniques and application security techniques with operation and network security measures (Hughes and Stytz, 2007).

This research supports two of the four thrusts: developing an algorithm to secure software and identifying metrics to stratify potency and cost. To satisfy these objectives, a series of investigative tasks was completed. First, we studied current software protection approaches for their merits and deficiencies, in addition to possible research directions based on their results. Second, we conducted analysis on techniques in fields, such as data cryptography and hardware anti-tampering, that share similar concerns regarding information protection. Third, we examined the uniqueness of the software operating environment to understand the specific challenges in a software-only solution. Furthermore, by understanding the nuances of the software operating environment, we can determine what approaches and techniques are feasibly adoptable or adaptable from other environments to function within the software environment. Finally, we test applications and evaluate them against developed criteria to examine efficacy of the new technique upon implementation. Applying the algorithm to actual software code creates a visual picture on the qualitative effectiveness and practicability of the model.

In summary, the delineated tasks approach the problem by looking outside the software domain, defining the uniqueness of the domain, and translating the techniques to work successfully back in the domain.

1.3 Scope and Methodology

Software protection is a broad topic covering a vast array of subjects including error handling, buffer checking, and mature software engineering. This research focuses on software protection as a means to prevent reverse-engineering of software algorithms. This research intends to describe a model for evaluating software protection, select metrics to evaluate security of within the model, develop benchmark programs to test against the model and develop an applicable software protection technique. The following research actions were required: review of established software protection models and related information security disciplines in literature, development of a protection algorithm, and demonstration of the developed algorithm using tools and techniques openly available to reverse-engineers.

We selected the virtual black-box model (Barak and others, 2001:2) and the obfuscating transform model (Collberg and others, 1997:2-7) as the two mainstream models for application protection; the software protection community recognizes both models as cornerstone works in this research topic. Of key interest are *how* each model describes the problem, *what* solutions are provided under each model, and *where* additional opportunities for refinement exist.

Data cryptography and hardware security, particularly field-programmable gate arrays (FPGAs), were selected as related fields of study, primarily for their emphasis on information security and analogous characteristics to the software protection problem. We examine both research areas to ascertain how security is characterized, what protective mechanisms are used, and how protective mechanisms function to secure information from adversaries.

1.4 Assumptions and Limitations

The finite number of test scenarios and generated benchmark programs is a limit on how well we can evaluate our proposed methodology. In addition, the obfuscation algorithm's "stamina" outside the established test environment remains to be validated by the cryptographic community following extensive real world assessment. In addition, known assumptions of the task include future validity of adapted techniques and computational power based on historical growth trends.

1.5 Preview

Chapter II provides an overview of the current research trends in theoretical and practical obfuscation. We also present terminology and key concepts commonly used within the field of software protection. Chapter III details the methodology for construction of the security model and protection technique in addition to the selection of metrics and design of benchmarks. Chapter IV reports on the results of the metrics, benchmarks, and techniques. In addition, we provide an objective evaluation of the proposed protection algorithm's efficacy based upon predefined criteria established in Chapter III. Chapter V summarizes the results of the research and highlights future research areas such as domains outside the software environment that may be able to benefit from this research.

II. Literature Review

2.1 Chapter Overview

This chapter examines research literature regarding information protection for adapting security techniques in designing and evaluating a general, efficient, and measurable software-only software protection method.

2.2 Objectives of Software Protection

The broad goal of software protection is to secure software or sensitive portions of the software from unauthorized analysis and tampering. Common applications for software protection exist in domains such as digital rights management (DRM), embedded systems, cryptographic software and mobile agents. For clarification, software protection is primarily concerned with confidentiality of software execution rather than confidentiality of the executed data (Yasinsac and McDonald, 2007:8; Loureiro and others, 2002:3).

2.3 Attacks on Software

In order to obtain these secrets from software, an adversary commonly employs two forms of attacks on software: analysis and tampering. Software analysis is the most crucial step to successful reverse engineering. Once an attacker understands how the internal algorithms function, the attacker can develop countermeasures, strip the algorithm for other programs, or extract information embedded in the algorithms, depending on his intent.

Tampering is purposeful modification of the software's behavior. A common example of tampering is applying a software patch to circumvent serial number copy protection mechanisms. Though discussed as a separate form of attack, analysis and tampering are closely related. At least a minimal amount of analysis is required prior to tampering; otherwise, the tampering attempt would be a series of random changes and unlikely to inflict the behavior desired by the adversary. Tampering is also commonly a part of analysis. As an extension of the previous copy protection example, an attacker can first perform an analysis on the serial number checker in order to build a patch to bypass the protection mechanism. During the analysis, an attacker may inject malicious code into the serial checker to better understand how it works through observation of its altered behavioral patterns (Cappaert and others, 2004:2).

While we emphasize counter-analysis approach to software protection in this research, we consider techniques regarding anti-tampering due to the tight coupling between the two forms of attack. We also discuss general security models to better understand the context in which the attacks operate.

2.4 Data Cryptography and the Black-box Model

Cryptography, the practice and study of hiding information, is a related field of study that involves keeping secrets and is useful as a framework in illustrating the key concepts of software protection.

In the basic cryptographic model, two parties, Alice and Bob, want to communicate secretly with one another. In this model, Alice and Bob implicitly trust each other and use a pre-shared key to encipher and decipher messages. Eve, an un-trusted party, can observe and intercept any messages going between Alice and Bob as pictured in Figure 2. For the purpose of consistency, Alice and Bob will always be trusted parties while Eve will also always be a non-trusted party when illustrating security concepts and models ("Alice and Bob," 2007).

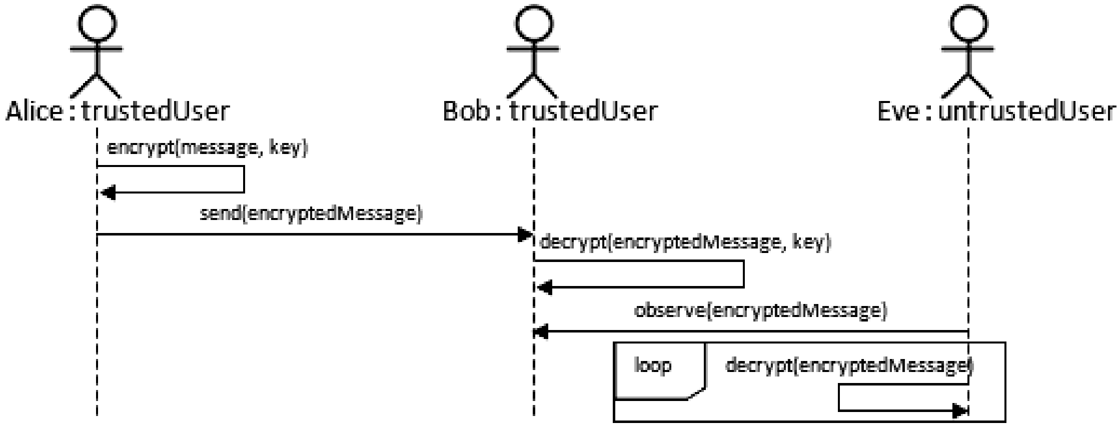


Figure 2. Basic Data Cryptography Model

This basic data cryptography model illustrates the concept of black-box security because the security strength is derived from the secret key used in the cryptographic primitive and not from hiding how the cryptographic primitive or how the underlying algorithm works. Eve has access to the same cryptographic algorithms that Alice and Bob uses to encrypt and decrypt the messages. However, without the secret key, Eve is unable to decrypt any of the messages.

Because Eve has full access to the algorithm behind the cryptographic primitive, she can conduct a black-box attack—a generation of any or all input/output (I/O) pairs. Then using statistical analysis, such as a frequency, linear or differential analysis, she can attempt to discern a predictable pattern between the input and output that may reveal information about the key or message. Strong encryption algorithms are designed to produce high entropy or randomness in the output to defeat statistical analysis (Preneel and others, 2003:13). In addition, strong encryption algorithms are also designed to make it too computationally expensive for Eve to enumerate all I/O pairs. Otherwise, it would be possible for Eve to first generate all I/O pairs, create a lookup table (LUT) of all possible I/O, and finally decrypt encoded messages from Alice or Bob against the LUT. The reason it is not currently possible for Eve to build a single LUT for a strong encryption scheme is because the computation needed to generate all I/O pairs is currently infeasible. For instance, in the case of the Advanced Encryption Standard (AES) which uses 128-bit keys and operates on messages in 128-bit blocks, generating the LUT would require computation and storage of 1.4 * 10^{79} bits (2^{128} keys * 2^{128} blocks * 128 bits/block) or 1.8 *10^{69} gigabytes (GB). The notable caveat to encryption strength is that what may be strong presently may not always be strong; the Data Encryption Standard (DES) was once considered strong, but has since been determined weak due to improvements in hardware that make it possible to crack the key in hours ("Data Encryption Standard," 2007).

It is important to note that encryption strength is not dependent on the secrecy of the algorithm. Cryptographic community approved algorithms such as AES have open-source implementations and undergo a standardized evaluation process. This adheres to one of Kerckhoff's cryptography principles where the strength of encryption should not be based on the obscurity of the encryption algorithms or the lack of understanding in how the algorithms operate ("Kerckhoff's principle," 2007). The open-sourced nature of cryptographic primitives also allows developers to implement the algorithms as designed in any cryptosystem as a measure to prevent improper implementation from inadvertently weakening the strength of the encryption algorithm.

2.5 Software Obfuscation and the White-Box Model

We make a few important observations about the black-box model that highlights characteristics of the white-box security model and the associated challenges with respect to software protection. The first observation is distinguishing what is secret and what the secret protects. In data cryptography, the pre-shared key is the secret used to hide the message as seemingly garbage data. In software protection, the software program or more specifically, the algorithm is the secret. Thus, something must prevent Eve from analyzing how the secret algorithm works. This leads to the second critical observation that Alice and Bob, at the two ends of the encryption/decryption process, are both trusted parties in data cryptography. In contrast, Alice must cooperate with Eve, the un-trusted party in the software protection scenario. Figure 3 illustrates the relationship between Alice and Eve in the standard software protection model to contrast their relationship in the basic cryptography model shown in Figure 2.

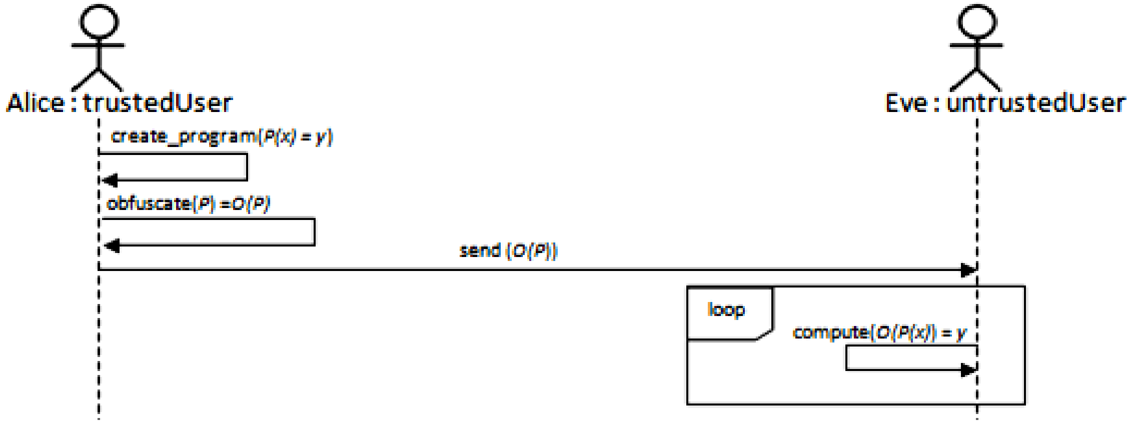


Figure 3. Software Obfuscation Model

A third observation shows the significance of this fact. In the traditional cryptographic model, Eve is not involved in the encryption and decryption process that occurs at the ends of the secret sharing process; she is, therefore, never privy to the secret that enables secure message transfer. Software protection, in contrast, requires her to either implicitly or explicitly know the secret because the executable code is the secret and she has the code. This yields yet another observation. An enciphered message passed in the open is useless except to someone with the proper decryption key. Software, however, must be at least interpretable by a compiler to execute; otherwise, there would be no reason for Alice to give Eve the program if Eve could not execute it. This, in turn, means that even after protection mechanisms are applied, the secret must remain interpretable on Eve's machine.

Table 1 is a summary of the distinctions between confidentiality of data (cryptography) and confidentiality of execution (software obfuscation). Of note, there exists a distinction between program structure, typically the source code, and program functionality, the algorithm's I/O. The distinctions between the data cryptography security model section illustrate that software protection does not conform well to the black-box security model alone.

Table 1. Differences Between Confidentiality of Data and Execution

Confidentiality	Data	Execution
Secret Object	key	program
Protected Object	message	program structure
Distributed or Public Object	encrypted message	program functionality
Actors privy to Secret Object	Alice, Bob	Alice
Actors accessing Protected Object	Alice, Bob	Eve
Actors denied Secret Object	Eve	Eve

Because Eve controls the operating environment of the software, the host environment can be used against the mechanisms protecting the application. Chow and others specifically states that in a white-box attack context, the adversary has three capabilities. First, the cryptographic software shares a host with the fully-privileged attack software with complete access to implementations of encryption algorithms. Because the attack software has the maximum ("admin") privileges allowed on the host, the target software cannot expect access rights within environment to offer any form of protection from the attack software. Second, dynamic execution of the target program is observable. This fact implies that Eve can monitor the execution of the target program's individual instructions to learn each instruction's behavior. Closely tied is the third capability where the adversary has complete view of the internal algorithm and can alter it at will. Combined with the second capability, the adversary is thus able to see the entire algorithm, run it line by line, and alter the instructions to analyze the target program. This capability allows the aforementioned, tampering in the process analysis, whereby Eve can inject code into the target program in order to gain understanding of how the program works (Chow and others, 2002:4).

Noting these capabilities, it is possible to understand why symmetric pre-shared key cryptographic primitives are strong in the black-box security model but vulnerable in the white-box security model. An example of this fact is the program encryption technique which encrypts a program and decrypts it during runtime. Once encrypted, the original program is no longer an executable program and therefore requires decryption prior to run-time. It follows that during execution, an unencrypted and thus, unprotected version of the program will exist in memory on the host environment. Under the white-box attack context, the unprotected version is subjected to analysis attacks on the host memory. Therefore, an attacker can bypass the code encryption by performing run-time analyses to directly observe or modify the unprotected code in memory as seen in Figure 4 (Collberg and others, 1997:5; Eilam, 2005:330).

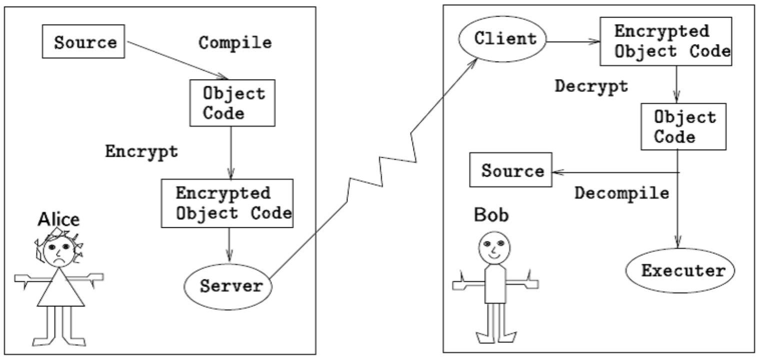


Figure 4. Program Encryption Model

Because the program encryption technique does not hold under the white-box security model, it is an insufficient means of software protection when used as the sole defense mechanism. For clarification, we are interested in protection encryption techniques that produce *executable* encrypted code in contrast to the approach in Figure 4 which produces *non-executable* encrypted code.

Giving the secret to the adversary is counter-intuitive to protecting it. Despite the difficulties faced in the white-box model, the challenges of the white-box model are a reality for software protection.

2.6 Theoretical Obfuscation and the Virtual Black-Box

In their seminal work on software obfuscation, Barak and others propose the virtual-black box (VBB) obfuscation model to characterize software security in the white-box model. VBB established three properties that laid the groundwork for most theoretical discussions about software obfuscation. Critical to these properties is the concept of the obfuscator (*O*), which accepts a program (*P*) such that a new obfuscated program (*O(P)*) is generated. First, *O(P)* must produce the same output as *P*—this is sometimes referred to as the function preserving property. Second, any computation by *O(P)* is performed with efficiency consistent with oracle access to *P*—this requires *O(P)* to run in the same computational time complexity, such as polynomial time, as *P*. Finally, the structure of *O(P)* must not provide Eve any useful information about the structure of *P*—this recognizes that software obfuscation is a white-box problem. Informally, an *O* that achieves all three properties means that Eve cannot discover more information from analyzing *O(P)* than from black-box analysis of *P* (Barak and others, 2001:2).

In their work, however, they construct theoretical arguments that show that it may be impossible to create a general purpose obfuscator that satisfies all three properties in the VBB model. They construct their argument by creating an efficiently calculated family of functions that provides more information about the function than just oracle access to the original code. Their "impossibility" results have spurned many research directions under software protection. First, because the impossibility result only precludes the existence of general obfuscators for all functions, there is a search for obfuscate-able families of functions. Second, there is a search for alternate security models that may better describe the software protection problem better than VBB. Barak and others concede in their work that the three properties making up VBB may not be a practical model (Barak and others, 2001:30; Hofheinz and others, 2007:218-222). New models modify one of the three properties in VBB or attempt to define the maximum obfuscation that can be achieved (Goldwasser and Rothblum, 2007:194). Other research directions include specific obfuscation techniques that, despite the impossibility results, are practically implementable as a means to slow down analysis attacks (Lynn and others, 2004:11).

2. 6 Applications

The following applications for software protection are presented to better understand the white-box security model and the software protection problem as well as motivation for this line of research. In addition, some applications illustrate why Alice may want to coordinate with an un-trusted party, Eve.

2.6.1 Cryptographic Software.

Cryptographic primitives are designed to operate within the black-box attack context and not the white-box attack context. This is a limitation in their use because the user must be concerned with where their software is installed. For instance, it is unwise to install unprotected cryptographic software on a random terminal because it could be under adversarial control and thus have plain-view of the secret symmetric or asymmetric keys entered into the software (Chow, 2002:252). If the cryptographic software could prevent the secret keys used in encryption from observation in the malicious environment, then users could perform secure message transfer in non-secure environments.

2.6.2 E-Commerce.

E-commerce applications often embed cryptographic keys and are a specific application of cryptographic software. More specifically, cryptography is used to ensure privacy, confidentiality and overall security through means such as e-certificates and digital signatures (Cappaert and others, 2004:31).

2.6.3 Digital Rights Management (DRM).

One of the biggest challenges with the prevalence and ease in spread of digital data is tracking and receiving revenue from creative media shared on the internet. Software protection measures are needed so that only those who have the rights to the media would be able to use it. Because the possibility of transferring the media file to a malicious buyer exists, software protection could help in deterring unauthorized duplication from those that may have legitimately purchased the item. In addition, specific protection techniques could watermark an item so that unauthorized copies are traceable to a unique user in order for the intellectual property owner to pursue legal action ("Digital Rights," 2007).

2.6.4 Software Agents.

Software agents are an abstract concept to describe pieces of software that acts on the behalf of the owner of the agent. There exists many possible uses for software agents but the common thread in every instance is that the agents may "travel" to potentially un-trusted environments while carrying sensitive information about the owner in order to perform its function. In this regard, the software protection allows the owner to deploy his agents to perform their task without fear that his private information may be extracted from them ("Software Agent," 2007; Sander and Tschudin, 1998a:2).

2.6.4.1 Shopping/Auction Agents.

In the case of a shopping agent, the owner deploys the agent to vendors but would not want any vendor to know the highest acceptable price would be for an item. Protecting the embedded thresholds on the agent would prevent the threat that a malicious vendor could artificially inflate the prices by extracting the owner's highest acceptable price. Fundamentally, the vendor would always want to know the buyer's strategy to produce the highest price acceptable by the buyer, while the buyer has an incentive to keep his information private to obtain the best price possible (Algesheimer and others, 2003:11).

2.6.4.1 Monitoring-and-Surveillance (MaS) Agents.

The typical use for a MaS agent is to observe and report on computing equipment or services. If an attack on a Supervisory Control and Data Acquisition (SCADA) system also compromised its monitoring MaS agent, it could impair authorities from detecting and responding to the attack on the SCADA system.

It is also possible to purposely deploy MaS agents to monitor adversarial networks and collect intelligence. When used as an intelligence asset, it is typically desirable that the agent not divulge the owner's identity in addition to the type of information the agent was tasked to record. Furthermore, the owner would want to prevent the adversary from learning how to create adversarial MaS agents through reverse-engineering of any captured MaS agents ("Software Agent," 2007).

2.6.4 Embedded Systems.

Embedded systems such as smart cards can benefit from software protection because of their highly distributive nature and their use in commercial applications. Smart cards often embed cryptographic keys but cannot afford the size increase in using full hardware anti-tampering techniques.

One notable hardware and DRM example is the Content Scramble System (CSS) distributed with DVD-players, DVD-drives, and the DVDs. CSS keys protect DVDs as a means to prevent unauthorized copying and distribution. Built-in CSS decryptors in DVD-players and DVD-drives decrypt the content on the DVDs to play them. Thus, this is a white-box security problem because the decryption mechanism and key are distributed to potentially malicious users. One malicious user, in 1999, successfully analyzed the CSS algorithm and produced a De-CSS algorithm, circumventing the protection mechanism ("Content Scramble System," 2007).

2.7 Software-based Protection

We focus on software-based approaches in this research because, in general, software-based techniques tend to be more compatible with existing systems. Since software-based protection techniques are typically modifications to the original program code, any hardware running the original code should also be able to run the software-protected code. This flexibility, in turn, offers an added layer of protection through frequent updates. In contrast, hardware techniques suffer from incompatibility with the original operating platform thereby incurring relatively higher costs when implementing protective mechanisms. Hardware characteristics such as power consumption and time analysis may also leak information about the original system through side-channels despite protective white-box measures (Cappaert and others, 2004:8).

It is useful, however, to examine traditional hardware protection approaches because hardware are physical items that must also be protected in both black-box and white-box attack contexts. An adversary could use a black-box approach and determine how the circuit works or develop another circuit which mimics the behavior of observed I/O-pairs (Christiansen, 2006:1). Anti-black-box methods are available to confuse the I/O-pairs, but are viewable in the white-box context. As a physical item, physical protection is an option in addition to the fact that hardware is more difficult to duplicate; this is an important factor because the amount of resources available to an adversary such as a college hacker, non-state actor, or nation-state intelligence agency can vary wildly. Therefore, strong physical anti-tampering measures can make the adversary assess the risk between conducting white-box attacks on the samples for information and destroying the finite number of samples.

Duplication of non-protected software code, conversely, is usually easy and only requires low-cost or even open-sourced tools. In addition, an adversary can safely tamper software with the capability to repeatedly revert to a backup copy if the working copy becomes unsalvageable. In summary, software-based protection must reflect the unique challenges of the software environment (Cappaert and others, 2004:8-9).

2.8 Current Solutions

We examine practical solutions and analyze their approach to the software protection problem.

2.8.1 Client-Server.

The effect of using a client-server solution is to force the adversary into a black-box attack context by storing a protected program on a protected server that is accessible remotely by the client. Thus, the adversary can only feed input into the client and receive output generated from the server. If the algorithm that needs protection is computationally intensive, then the server must bear the computational load. Improvements to this approach offload as much of the non-sensitive on the client side. However, there is a limit to the amount of the processing of the protected function computable on the client because removing access to the critical portion of the code is the sole protective measure. In addition, there is a high reliance on the interactive communication between the server and client that may not be possible or desirable for certain applications such as intelligence MaS agents. The challenges presented in the white-box attack context are averted because the fundamental concept in this security technique is protection of the server rather than protection of the software (Cappaert and others, 2004:10).

2.8.2 Code Encryption.

The white-box security model section, illustrated in Figure 4, shows how standard data encryption schemes applied directly to code is typically insufficient as a software protection. In a work on mobile agents, Sander and Tschudin propose a scheme that uses homomorphic functions as a specific form of code encryption that may work in the white-box attack context. However, few homomorphic functions exist and are themselves a subject of research with few available implementations (Sander and Tschudin, 1998a:9-10).

2.8.3 Obfuscating Transforms.

Collberg and others' works are examples of research emphasizing implementable techniques for obfuscation to provide practical levels of security. The basic appeal in their work is that like data encryption, obfuscation is a process that delays the adversary from achieving his goal (Collberg and others, 2002:738). Data encryption uses math and complexity theory to estimate the theoretical maximum amount of time an adversary would need to successfully conduct a brute-force attack, where an adversary generates all possible combinations to decipher the hidden message. Brute-force attacks use the entire input space and therefore guarantee success by definition. However, secure data cryptographic schemes ensure that a full brute-force attack would take an unpractical amount of time to complete with current computer processing capabilities. Effectively, cryptography delays the adversary from knowing the encrypted information until it is of negligible value or until newer technology and faster technology is available to execute the brute-force attack. In either case, cryptography does not claim impregnability, but merely a mathematically measured amount of delay. Similarly, Collberg and others propose that given enough time, a competent reverse engineer can reverse any software program (Collberg and others, 1997:1-29). According to their current works, however, it is not possible to mathematically derive the amount of delay on analysis. Instead, their foundational work on the taxonomy of obfuscation techniques focuses on classifying the types of the obfuscating transform mechanisms and their relative security to each other. A quality metric composed from four properties: obscurity, resilience, cost and stealth. An abstract scale for each property provides a relative measure on the approximation of strength of one technique against one other.

Generally, their proposed obfuscation techniques provide white-box protection using conventionally poor software engineering techniques. Therefore, the metrics proposed to measure obfuscation are derived from sound software engineering principles. The quality of each technique is assessed according to four core properties (Collberg and others, 1997:7).

2.8.3.1 Obscurity.

Obscurity is a measure of complexity or unintelligible-ness of *P* or more importantly, *O(P).* This metric is conceded by the authors to be vague since it is an attempt to quantify cognitive ability using the q-property, a loosely defined property that makes *O(P)* more complex than *P*. Thus, the reverse-engineering attempt of *O(P)* should be more time consuming than reversing attempt of *P* if the *q*-property is greater in *O(P)* (Collberg and others, 1997:7; 2002:4).

2.8.3.2 Resilience.

Resilience is similar to obscurity. While obscurity is based on confusing a human's understanding of *O(P)*, resilience is based on a technique's ability to defeat automatic machine de-obfuscators. Because de-obfuscators must first be programmed by a person, resilience is a combined measurement of both the time it takes a programmer to construct the de-obfuscator and the time it takes the de-obfuscator to reduce the obscurity of *O(P)*. A de-obfuscator's effort is often described in classical complexity terms such as polynomial-time or exponential-time (Collberg and others, 1997:8; 2002:4).

2.8.3.3 Cost.

Cost is the amount of execution-time and memory space incurred by the obfuscation technique and often described in classical complexity terms and bytes, respectively. If a technique is prohibitively expensive, it would be of little practical value regardless of the increase in security because *O(P)* may no longer meet non-security related criteria required in *P* (Collberg and others, 1997:9; 2002:4).

2.8.3.4 Stealth.

Stealth is used in a special application of obfuscation for watermarking. It is a property that measures the closeness in statistical similarities between the original code in *P* and *O(P)*. Software protection techniques that use obfuscating transforms as a watermark would attempt to increase this property to avoid detection (Collberg and others, 2002:738).

These four properties describe the obfuscating transform model because they were conceived as ways to evaluate the various practical obfuscation implementations through transformation of source code. Collberg and other's work falls short on meeting the scrutiny of the cryptographic community but makes significant strides in achieving positive results for obfuscation that are capable of practical implementation today (Collberg and others, 2002:738).

2.8.4 Hardware Techniques.

Field-programmable gate arrays (FPGAs) blur the distinction between hardware and software. FPGAs' logic are programmable post-production and have the advantage of having shorter time-to-market costs due to lower non-recurring engineering costs and easier patching of bug fixes. The white-box security model applies to FPGAs because the FPGA can fall into adversarial hands. The logical programming of the FPGA is the proprietary secret that must require protection and can be protected through software protection means rather than through hardware anti-tampering measures alone (Vahid, 2007:106).

Hardware has the advantage of being a physical device. AFRL/SPI has explored solutions such as the embedding of a small explosive payload on top of FPGA boards that upon tampering destroy the board. A software equivalent might be a malware or virus embedded into the software that activates upon detection of a dissassember on debugger on the host environment or upon knowing that it is running within a virtual machine environment. This type of protection focuses therefore on breaking the specific analysis tools rather than the adversary. While this is a valid approach and an active avenue of research, it produces an arms race of techniques where analysis tools and anti-analysis tool techniques are incrementally patched to compensate for new developments (Eilam, 2005:328-329; Travis, 2001).

2.9 Bridging Theory and Practice

This research attempts to model a software protection model that is practically implementable with quantifiable metrics to bridge the gap between theoretical obfuscation and practical implementation. Chapter III details the methodology used in the proposed model's architecture and the experiments we performed to examine the model.

III. Methodology

3. 1 Chapter Overview

This chapter describes the research experiments, beginning with presentation of the research goal and approach. We describe the test boundaries and services followed by the metrics we use to interpret the results of the experiments. Finally, we present the experimental design to describe how the set of experiments answers questions posed in the problem definition.

3.2 Problem Definition

Three attributes generally characterize software protection techniques: applicability, efficiency, and security. The Holy Grail of software techniques would be one that is general in application, secure in implementation, and efficient in execution. Thus far, however, research in theoretical obfuscation has yielded positive results that are provably secure but applicable for only specific functional families (Lynn and others, 2004:11).

Practical obfuscation approaches use software engineering metrics that are easily applicable to existing software. Security metrics, however, remains a research area because breaking software protection techniques is in part art and in part science. Software engineering metrics were conceived as metrics to gauge the likelihood of coding errors, not as security metrics. Thus, the software engineering derived metrics and corresponding properties for evaluating software obfuscation are understandably weaker than metrics used by traditional cryptographers in evaluating cryptographic algorithms. This suggests obfuscation techniques with more generically quantifiable metrics, independent of cognitive ability, would appeal to both experts in practical and theoretical obfuscation. Table 2 presents software engineering metrics in white (Collberg and others, 1997:8) and cryptographic metrics in grey for comparison ("National Institute," 2001). We note that the software engineering metrics are traditionally used to assess program complexity where an increase in a metric indicates increased overall complexity while the cryptography metrics are used to indicate the randomness of a bit string produced by encryption algorithms or pseudo-random number generators. A bit string with high randomness means that it is difficult to guess the outcome of a bit with greater than 50% accuracy.

Table 2. Measures in Software Engineering and Cryptography

Metric	Short description
Program Length	Number of operators and operands in P
Cyclomatic Complexity	Number of predicates in functions
Nesting Complexity	Number of nesting level of conditionals
Data Flow Complexity	Number of inter-basic block variables
Fan-in/out Complexity	Number of formal parameters and/or global variables
Data Structure Complexity	Number of fields, size, type of static data structures
Object-Orientated Complexity	Number of depth, inheritance, methods, coupling
Frequency	Proportion of 0's and 1's
Frequency Within a Block	Proportion of 0's and 1's within multiple sequences
Longest Runs of 1's in a Block	Length of uninterrupted sequence of 1's
Runs of 0's and 1's	Number of uninterrupted runs of 0's and 1's
Cumulative Sum	Sum of partial sequences after mapping (0,1) to (-1,1)
Random Excursions	Number of cumulative sum cycles with 0 sum
Random Excursions Variant	Number of sums within cumulative sum cycles

This research proposes a software-only approach using compositional function tables (CFT) and embedded symmetric key cryptography to produce functional entropy on a small scale for the protection of deterministic functions. Functional tables are the perfect white-box because only the input/output pairs are made available. Thus, a function table provides just the black-box information. By replacing a deterministic function with a function table, we strip the structural implementation of the function to prevent white-box analysis by the adversary.

The objective of this research is to examine the effectiveness of symmetric key cryptography and CFTs as a software-only protection technique. Of primary interest is how well this approach quantifies obfuscation strength with measures and metrics consistent with ones used in traditional data cryptography. In addition, this research proposes a set of benchmark programs to demonstrate this approach and may be useful in determining effectiveness of current and future obfuscation techniques. Finally, we evaluate the generality and efficiency of the CFT approach.

We select Java programs and methods to implement our experiments because decompiled Java code of unprotected functions is very similar to the original Java source code providing a greater contrast between decompilations of protected and unprotected code. Compiled Java code is also more understandable because it compiles into a well-documented bytecode format which retains internal symbolic information, such as class names, that help the adversary and Java de-compilers, such as Mocha, reconstruct the original source code and logic. In contrast, C/C++ code compiles into microprocessor instructions that contain less information about the original code and therefore gives less information to an adversary. Popular C/C++ reverse-engineering tools, such as OllyDbg and IdaPro, are dissassmblers which generates the assembly level instructions, rather than the original source code making qualitative comparison against original source code more difficult. Furthermore, Java is a popular choice for web applications that often execute on un-trusted environments. For these reasons, we chose Java as the language to implement this research's experiments (Travis, 2000; Torri and others, 2007).

3.3 Alternate Obfuscation Model

The VBB model indisputably describes the ideal criteria for software obfuscation. However, theoretical research has shown that this ideal model is impractical. Therefore, an alternative model is necessary to describe a set of obfuscation criteria that does not lead to the same impossibility results produced by Barak and others.

The three criteria established by the VBB model state that an obfuscated version must preserve functionality of the original, perform in equivalent time to the original, and reveal no information about the original that cannot be obtained by having only black-box access to the original. This research examines a model that removes the first criterion: function preservation. Removing this criterion is clearly a weakening of the VBB model, but in turn shelters this new model from the established impossibility results. Of note, this alternate model clearly distinguishes between the structure of program (white-box information) and the function of the program (black-box information) to reflect our observations in Table 1 where we identified differences between the data cryptography model and the general software obfuscation model. McDonald and Yasinsac propose that obfuscation, at best, protects the structure, protects the functional relationship, or protects both naming this the intent protection model (McDonald and Yasinsac, 2007:2-3).

Other research works also support black-box protection of the function output as a means to obfuscate the white-box structure. Sander and Tschudin propose a protocol for computing with encrypted functions (CEF) under the premise that reversing the underlying proprietary functions generally more useful than full reversal of the program (Sander and Tschudin, 1998b:2). Loureiro and others uses a Boolean equation set representation of the function table approach with the McEliece asymmetric cryptographic algorithm which encrypts the output as an obfuscation technique (Loureiro and others, 2002:4). Chow and others also use combinations of function tables to integrate their white-box version of the AES algorithm to protect other functions (Chow and others, 2002:252). These works all emphasize the need to modify the functionality of the original function as part of an obfuscation technique. We noted that the output is unusable until it is converted back to some usable form, which is usually done on a trusted environment. Figure 5 graphically illustrates this intent protection model for comparison with the VBB standard obfuscation model in Figure 3 and the standard cryptography model in Figure 2.

While it appears that this is the client-server model, there is a key distinction. Traditional client-server hides the proprietary algorithm on the server side forcing the server to bear the computational load. In contrast, the objective of the partial client-server model is to safely offload the computational load onto the client. For example, a MaS agent, such as the ones described in the previous chapter, can perform secure computations within an un-trusted execution environment and then send information back to the issuer.

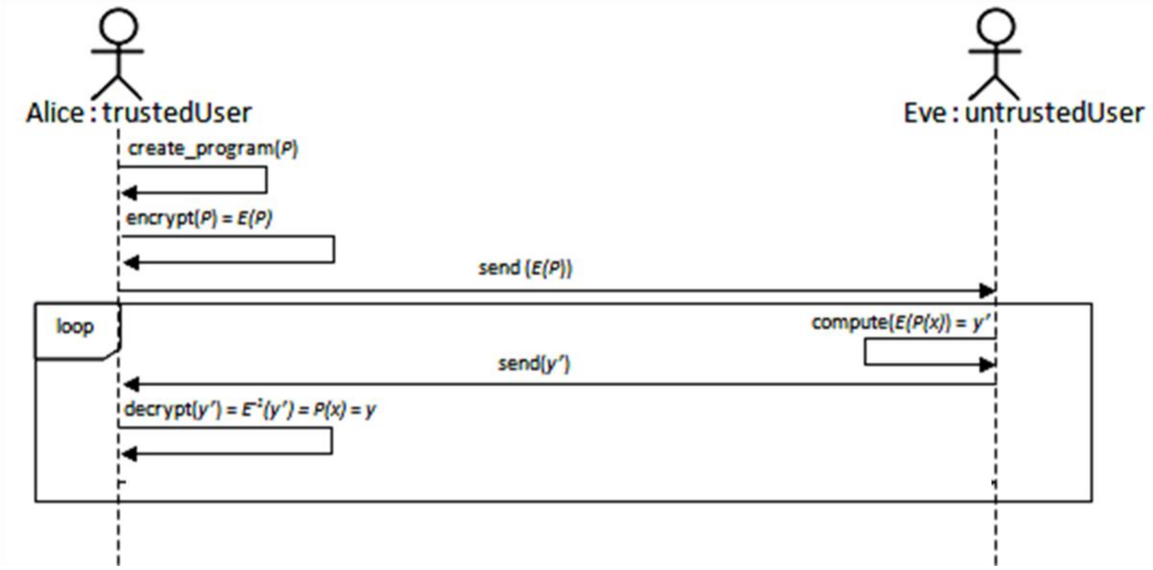


Figure 5. Intent Protection Obfuscation Model

Intent protection weakens the first criterion (functional preservation) of the VBB model. However, by providing functional confidentiality, it may be possible to strengthen protection overall through the third VBB criterion (structural confidentiality). Because VBB requires functional preservation, analysis of black-box information in the original and obfuscated version of the function may allow the adversary to extrapolate the white-box information. This is acceptable, though unintuitive, in the VBB model, especially when we know that adversaries use a combination of black-box and white-box attacks. Conversely, if it is acceptable within an obfuscation model to change the functionality of the obfuscated program, then it is possible to apply techniques that prevent deduction of white-box information through black-box analysis.

We thus revisit data cryptographic techniques since their primary function prevents black-box analysis. We note that any encryption of the output is still in a white-box attackable environment and thus methods for white-box encryption require examination. Figure 6 illustrates the data cryptographic model on the left with the intent protection model on the right for comparison.

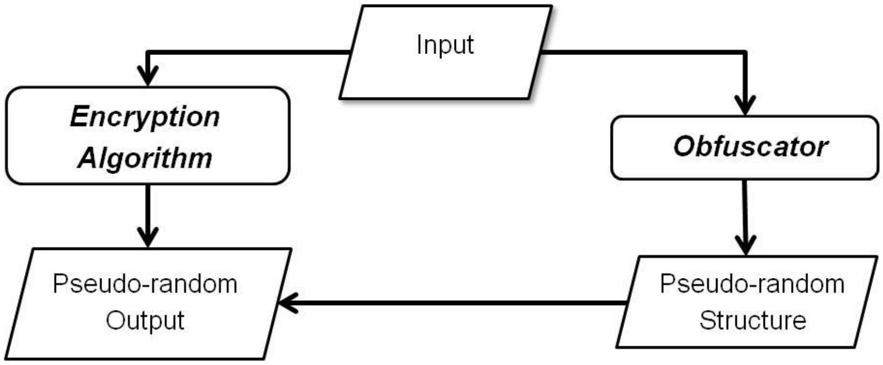


Figure 6. Data cryptography (left) and Intent Protection (right)

As stated previously, we divide a function into its functional behavior and its structure. While an encryption only makes the output appear as random output, we postulate that an obfuscator must also protect the white-box information. We could achieve this by either removing structural information or by emulating structural randomness. Thus, this research examines the input/output produced by random programs for comparison with similar sized functions to gain understanding on the relationship between the randomness in a program structure with the randomness in the output. To the best knowledge of this research, the relationship between random structures and corresponding output characteristics is unknown. If obfuscation is analogous to cryptography, then we can make the same analytical comparisons on the output. For instance, in order to gauge how well an encryption produces a pseudo-random output, it must exhibit characteristics comparable to a truly random sequence. The National Institute of Standards and Technology (NIST) published a list of established metrics that can empirically determine how closely a sequence exhibits randomness ("National Institute," 2001). Methods to accurately assess the level of randomness of function or program structure are, at this point, unknown and a reason why it is difficult to practically evaluate practical obfuscation techniques under VBB model's security test posed by the third criterion.

This research postulates that any program generated by randomly selecting bit manipulations between the input and output is a random program. Specific implementation details on how this research creates random programs are in the experimental section. By creating randomly generated programs, it is possible to examine their output using statistical measures. If random programs generate non-random output, then it is possible that obfuscation through randomization of structure is sufficient because the output does not correlate strongly to the structure. An indicator of this would be a large set of random programs that produce the same output pattern. If random programs tend to generate random output, then any program, original or obfuscated, that does not produce random output may indicate that a strong relationship between black-box patterns and white-box structure exists. Therefore, even if randomness is induced into the structure, it may never be sufficiently enough due to the predictability of the output. Security then requires a mechanism to produce randomness in the output which intent protection model supports (Hofheinz and others, 2007:17; Algesheimer and others, 2003:5).

Figure 7 illustrates the comparisons made in this research under the intent protection model relative to the comparisons made in the VBB model. In summary, the obfuscation community has yet to agree on how to make structure comparisons for white-box security. Without consensus on the structural security measure, it is difficult for practical obfuscation techniques to claim meeting the VBB security test as shown by the leftmost arrow. Thus, we propose the random program model, where *O(P)* is made to functionally and structurally resemble random programs (P_R), as a derivation of the random oracle model in cryptography (Bellare and Rogaway, 1995). Constructing P_R serves as an intermediate step in understanding and evaluating function structure and output patterns. We can use the results to develop techniques so that *O(P)* exhibits both functional and structural characteristics of P_R.

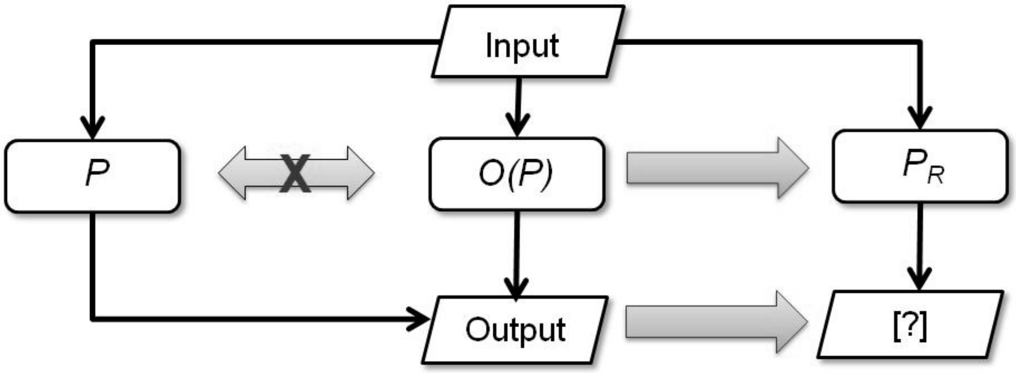


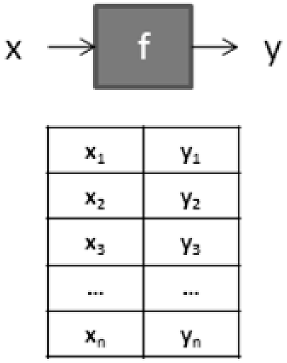
Figure 7. Obfuscation and Random Programs

Canetti and others prove in their work that work that techniques secure in the random based methodology may be insecure in implementation. However, we note that the cryptographic community uses the random oracle because the standard cryptography model based solely on complexity measures is difficult to prove. Therefore, our appeal to randomness is primarily to establish a sanity check on obfuscation approaches, as recommended by Canetti and others in their conclusions, in absence of a stronger security model (Canetti and others, 2006).

3. 4 Function Tables

We examine the removal of white-box information as an obfuscation measure. For this approach, a function table, which is a list of input/output mappings, is used. Every deterministic algorithm produces a function table. As stated previously, a function table hides all white-box information making it a perfect white-box.

Correspondingly, a function table is also an atomic function; it is not possible to observe, insert, divide or otherwise tamper with the instructions that generate the input/output pairs within the function (Yasinsac and McDonald, 2007:2-10). This concept is illustrated in Figure 8 using a generic function, $f\colon \{0,1\}^x \rightarrow \{0,1\}^y$ where x_n inputs map to y_n outputs.

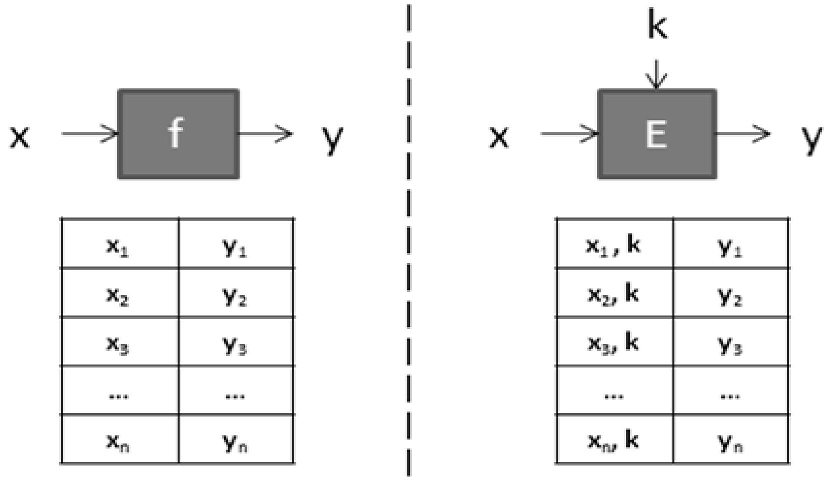


x_1	y_1
x_2	y_2
x_3	y_3
...	...
x_n	y_n

Figure 8. A Generic Function and its Function Table Representation

We note that an infinite amount of functions can produce the same function table. For instance, the same table is produced by the deterministic function, *f(2x),* is the same as *f(x+x)* and *f(x<<1)*. However, it is not possible to tell from the function table *alone* whether we used an addition, multiplication, or left shift operation in this simple example.

A generic encryption function, *E*, is also a function that takes an input and generates an output. Only a few characteristics distinguish an encryption function from a generic function. First, encryption functions have the property where the input and output generated are the same size. Second, the relationship of input and output for a particular encryption is identified by an key, $\{0,1\}^k$. The relationship between *f, E, x, y* and *k* are shown in Figure 9.



x_1	y_1
x_2	y_2
x_3	y_3
...	...
x_n	y_n

x_1 , k	y_1
x_2, k	y_2
x_3, k	y_3
...	...
x_n, k	y_n

Figure 9. Generic Function (left) and Generic Encryption Function (right)

The output of a strong cryptographic function is designed to exhibit randomness such that it is not possible to guess the output from previous input or input patterns. Thus, the functional table of an encryption function operates like a random oracle—a black-box that responds with a uniformly random response. Because a truly random oracle only exists as a theoretical mathematical construct, an encryption function is actually generating pseudo-randomness.

Strong cryptographic algorithms, such as AES, must exhibit sufficient empirical randomness in the output to make it resistant to linear or differential analysis attacks. We note that empirical security does not offer perfect secrecy such as a one-time pad ("National Institute," 2001; Jorstad, 1997). However, it is of sufficient practical strength that the National Security Agency (NSA) approved the AES algorithm for encrypting documents up to Secret classification. The randomness, or more accurately, pseudo-randomness is predictable only by knowing the cryptographic algorithm and the secret shared key. Due to the atomic property of function tables, all cryptographic implementation details, such as the secret key, are embedded within the table. Therefore, the adversary cannot view the key nor extract it through static or dynamic analysis once we make the table. For clarification, we list k with x_n in Figure 9 because it is a required input for the encryption function to generate y_n. Once we construct the table, however, an encryption function table would only include x_n and y_n.

A non-extractable key has significant security implications because it is now possible for symmetric key cryptographic primitives to securely operate in malicious environments. Thus, this research proposes to use atomic properties of the function table to white-box protect cryptographic primitives.

3.5 Function Composition with Function Tables

Functional composition, $(f \circ g)$ or $f(g(x))$, is also an atomic operation. Because the composition of two atomic functions is also an atomic function, it is not possible to find a seam between the two composed functions. We expect this because the seam is an implementation detail inherently hidden within the produced function table. As a result, a composite function table (CFT) protects the white-box information from each component functions in addition to the seam between them.

Composition on two functions is possible if the output of the first function is a subset of the input of the second function. Figure 10 is an illustration of a composite function of f and g. We see that the CFT of *(f o g)* masks f and *g's* individual input/output relationship; x_m thus directly relates to y_n with the intermediate information, y_m and $x_{n,}$ embedded in the CFT.

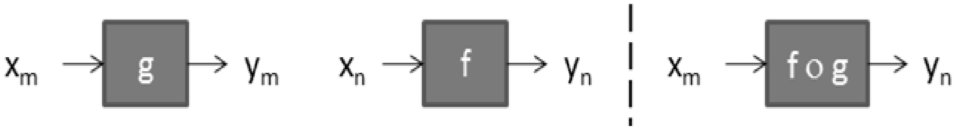


Figure 10. Function composition of f and g where $y_m \in x_n$

Without the intermediate information, it is difficult for an adversary to divide the table back into separate tables for f and g. To break apart the composition, he must compute all function tables equivalent to the input/output-size of the CFT; this is a super-exponential process. While not prohibitive for small inputs such as n=16 where $n^n = 2^{64}$, it is computationally infeasible for larger inputs such as when n>32 and where n^n>=2^{160}. Even for smaller inputs, the adversary must test combinations of generated function tables which is an n-factorial process (Yasinsac and McDonald, 2007:10).

3.5.1 Function + Encryption (F+E) Function Table.

The atomic properties of CFTs are the fundamental basis for our approach of securing a generic deterministic function, *f*, with an encryption function, *E*.

Encryption is essentially a recoverable semantic translation of some input. We use an one-bit input, one-bit output function to illustrate the CFT approach. There are exactly four semantic transformations, or behaviors, available to a function that operates on one bit which we list in Table 3. Within the table, we use a Boolean gate as the encryption function with another single bit as the key.

Table 3. List of Semantic Transformations and Sample Input and Output

Semantic Transformations	Candidate Encryption	x	y
1. Preserve the input	y = OR(x, 0)	0, 1, 1, 0	0, 1, 1, 0
2. Flip the input	y = XOR(x, 1)	0, 1, 1, 0	1, 0, 0, 1
3. Flip 1's, preserve 0's	y = AND(x, 0)	0, 1, 1, 0	0, 0, 0, 0
4. Flip 0's, preserve 1's	y = OR(x, 1)	0, 1, 1, 0	1, 1, 1, 1

The choice between the first and second semantic transformation is the best obfuscation possible for this trivial one-bit case because the adversary has, at best, a 50% chance of guessing whether we used the first or second transformation. The third and fourth semantic transformations are unsuitable candidates as an encryption function because they produce irrecoverable output. We note that the candidate encryptions in Table 3 are not the only possible implementation of the semantic transformation; an infinite number of functions can produce the same transformation. In summary, by selecting and composing a function table with an encryption table, the produced (f o E) CFT embeds within it, the input/output of f, the input/output of E, the k used in E, and the seam between f and E.

Popular encryptions, such as DES and AES, are recoverable semantic transformations whose behavior and recoverability is determined by the key and mode of operation. We chose electronic code book (ECB) as the mode of operation due to the necessity in enumerating input/output pairs for the encryption function table.

The primary weakness of ECB is that it does not hide patterns; identical plaintext blocks encrypt into identical ciphertext blocks. However, other modes of operation such as cipher-block chaining (CBC), cipher feedback mode (CFB) and others require the cryptographic primitive to have sense of causality; to generate the current ciphertext block requires information from a previous iteration. In other cases, a keystream, generated from the original key is used instead of the key for every new block of ciphertext. In either instance, the resulting functional tables would be super-exponential because all sequences need enumeration and mapping in order to fully construct the table. A single functional table can encapsulate the input/output mapping for an ECB operating encryption more compactly than for any other mode of operation ("Block Cipher," 2007).

To compensate for the leakage of input patterns, we suggest using padding schemes on the output of the first function before composing it with the second function. The most essential requirement for padding is that the receiver can distinguish the padding from the cipher text. Since decryption functional tables are a mirror of the encryption tables, any padding can satisfy this basic requirement because the padding that was included in the encryption table is reflected in the decryption table. We examine secure padding, such as RSA-OAEP (Optimal Asymmetric Encryption padding) designed to achieve statistically distributed 2^n output. We note, however, that we use random bijection tables rather than the RSA-OAEP to provide padding in this research so we have control over the size of the input and output space which may be smaller than the padding provided RSA-OAEP. Patterns in functional output carried through by the encryption algorithm in ECB is thus masked by the padding forcing the adversary to perform the super-exponential enumeration black-box attack plus the factorial process for combining function tables.

ECB has another weakness; it does not provide means for data integrity protection. This means that it is possible for an adversary to conduct replay attacks where the adversary interrupts the normal input process by inserting an input recorded previously. Though we note this is problematic for communication protocols, our current focus is using the encryption to protect the structure and functionality of the first function in the CFT, rather than protecting the system against communication protocol attacks.

3.6 Output Recovery

A decryption table is simply a reverse lookup of the encryption table and automatically created upon creation of the encryption table which is an exponential process. Because we perform decryption on the trusted side in the intent protection model, decryption does not need to be in the form of a function table because as long as the symmetric key is known, we can use a standard decryption primitive. This allows an issuer to distribute an obfuscated program using CFT and let a different trusted source receive and use the encrypted information without sending the entire enumerated table. This is beneficial if the bandwidth restricts sending the encryption table which may be of significant size.

It is important to note that the primary purpose of this approach is to allow secure computation on remote applications. Figure 11 illustrates how intent protection by partial client-server is distinct from the classical client-server model. The classical client-server model achieves function protection by removing all sensitive calculations from the remote application and running them only within a trusted environment that in turn places the burden of computation completely on the server side.

We can either directly decrypt and use the output or send it back to the remote application as necessary. However, if the output is passed directly back to an un-trusted user, an adversary could again have access to the original black-box information that was intent protected. In cases where the output needs to be used directly by the un-trusted user, the protected function must also be represented by a composition where *h(x) = f(g(x))*. We gain the most benefit if the computation for *g* is much greater than *f* because *g* is the portion of the function distributed as seen in Figure 12. In this configuration, the adversary has the output of the original function but does not have the full white-box structure of *h* with just (*g o E*).

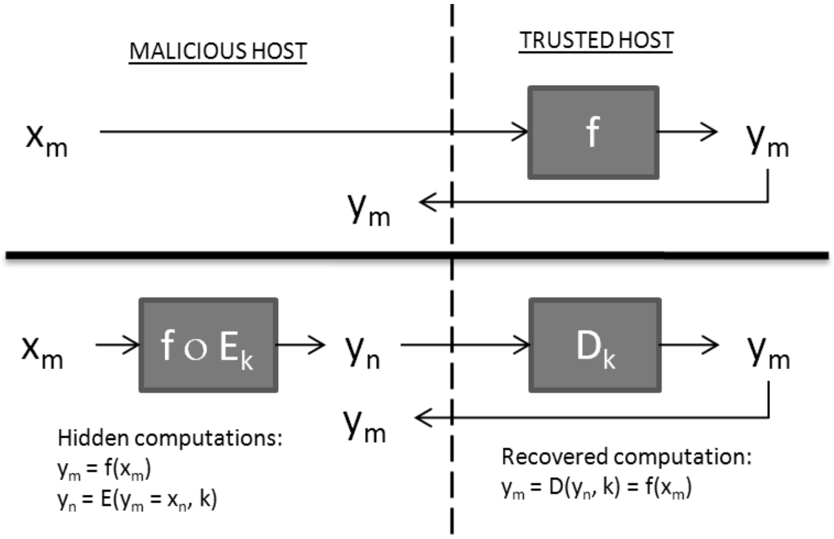


Figure 11. Classical Client-server (top) and Partial Client-server (bottom)

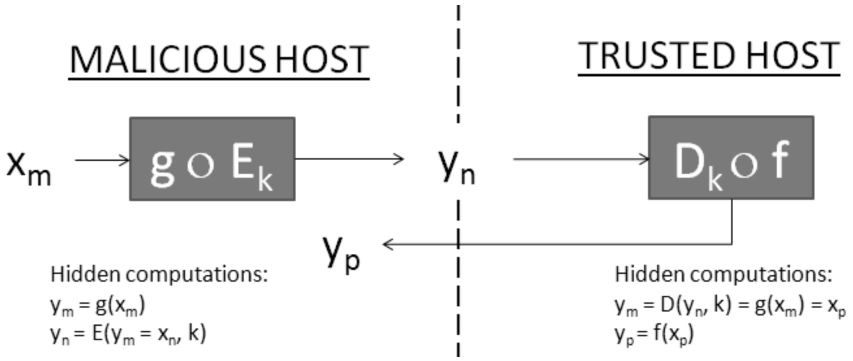


Figure 12. Output Recovery for $h(x_m) = f(g(x_m)) = y_p$

Implementation by partial client-server incurs bandwidth requirements that may become a bottleneck. However, this deployment does illustrate how it is potentially possible to securely offload intensive process to the remote applications. To eliminate the bandwidth bottleneck, it is possible to use two functional compositions in conjunction to produce a watermark that identifies the author of the function. This method also requires that the protected function is divisible into a function composition. However, because the malicious hose performs the entire computational load, the computational complexity differences between *g* and *f* are not significant factors in contrast to the partial client-server implementation. Figure 13 illustrates how atomic and individually useless functional tables can be used to secure a function, *f(g(x))*, and produce a unique watermark through symmetric keys.

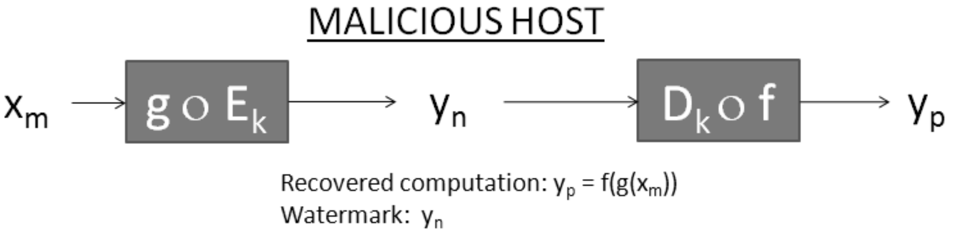


Figure 13. Watermarking the Composite Function $y_p = f(g(x_m))$ with Watermark y_n

In this configuration, functions *f, g,* E_k, and D_k, are secure through functional composition and generates the desired output without needing to contact a trusted source. Atomic properties of function composition protect the embedded keys in E_k, and D_k. The watermark at y_n, is a reproducible set of pseudorandom sequences generated with *E* and *k*. Even though it is observable by the adversary, the x_m/y_n and y_n/y_p input/output pairs do not reveal structural information about the individual functions *f* and *g*. Thus, even if the adversary knew that AES was the encryption algorithm used, he is forced to enumerate 2^{128} function tables for each key. It would be statistically improbable, given the empirically random pattern, for the adversary to find another function that generates the exact same pattern of randomness despite knowing that a theoretical infinite number of functions produce the same function table.

Software watermarking, like obfuscation, is a new research subject. Thomborson defines three aspects of a watermarking system. First, there must be an embedder to embed the watermark into the program. Second, an extractor must be available to extract the watermark from the program. Finally, an attack set must be constructed to understand ways an attack can modify the watermarked program; an attack set is the set of all attacks where the attacker can disable the extractor from removing the watermark (Thomborson, 2007).

For CFTs, the embedder and extractor is the *E, D* pair which is easily composed or removed from the protected function. The attack set is more difficult to model because various ways exist for the adversary can attack this scheme. Adding or deleting entries from the CFTs would not effectively destroy the watermark because the watermarking set of sequences would still exist in the maliciously modified program. Composing another function before the encryption CFT or after the decryption CFT would yield the adversary an altered program that retains the watermark of the original. Composing tables after the encryption CFT or before the decryption CFT would break the watermarking sequence, but most likely break the functionality as well because the adversary does not know f and g.

It would be statistically improbable, due to the empirical randomness generated by the cryptographic algorithm, for the adversary to effectively tamper with the logic by replacing just one of the functional tables. However, since this research works with programs and functions of enumerate-able size, it is possible to replace the entire structure of the two CFTS with a single LUT thereby removing all structural and intermediary white-box information which includes the watermark.

An alternative to the function table is its representation as a set of Boolean equations. Boolean equations sets (BES) are also two dimensional representation of input and output that reveals no structural information. For small input ranges, it may also be feasible to logically reduce the CFT into a BES by using the Quine-McCluskey or Espresso algorithms. By providing only the minimized equations, we force the adversary to conduct an n input-sized black-box attack to recover the functional table. However, a black-box attack is only O(2^n) complexity where Quine-McCluskey is O($3^n/n$) meaning the issuer does significantly more work compared to the amount of work required to undo the protective measure ("Quine," 2007). Though computationally costly for the issuer, BES configurations remain an option when constructing two dimensional structures for a function. Table 4 illustrates a pseudo-coded Java method represented as an equivalent functional table and BES.

Table 4. Pseudo-code for a Conventional and Unprotected Deterministic Function

public genericFunc(int a) {
….
int result = operations with a
return result;
}
	public genericFunc (int a) {
	return lookupTable(a);
	}
	lookupTable = [# # # … #]
		public genericFunc(int a) {
		result[0] = (a[1] & a[2]) …
		result[1] = (a[0] & a[3])…
		…
		return result;
		}

In the implementation where we deploy an encryption and decryption CFT accessible to the adversary, using BES representations of each would require the adversary to perform two black-box attacks which measurably delays the adversary's ability to destroy the watermark by replacing the encryption and decryption BES with one LUT. The logic minimization problem is NP-hard, but this limitation should not be prohibitive because we intend this approach for functions with small or bounded inputs ("Quine," 2007).

3.7 Developing an Implementation

Generating function tables from an atomic function requires an enumeration of the desired range of inputs to obtain the I/O pair. This is an exponential process, but only needs to be done once and done on high-end machines operating in trusted environments.

Because the function tables are direct lookups, we can optimize performance for the specific environment such as mobile applications that may have limited processing power. Applications commonly used function tables for mathematical functions such as sine or co-sine when computational capabilities to calculate floating point operations were unavailable ("Lookup," 2007). However, the trade-off comes in storage; the size of the function table is (2^n *m / 8) bytes of memory where n is the number of inputs and m is the output size in bits. For even 24-bit inputs, full enumeration and storage would require 268 MB. If an AES encryption is used, any table encapsulating just 21-bits of input would require approximately 4 GB since each output is 128-bits. Exponential memory requirements is a factor to consider if the end application using function tables must be deployed via networks with limited bandwidth.

One benefits of using LUT as an implementation is the understandability of its structure. As long as we populate the table with the correct values, table look ups are very low in complexity according to the software engineering metrics in Table 2. Thus, by using software engineering metrics, the CFT approach rates very low in security when an attack is theoretically n-factorial in computational complexity. A measurably secure approach that is low in software complexity is desirable because an issuer can focus on maintaining the complete program rather than the producing errors when adding in the obfuscation mechanism.

3.8 Approach

We first examine the black-box characteristics of random programs. We emulate random programs by generating combinational circuits with random structure following the example set forth by the IEEE International Symposium on Circuits and Systems 85 (ISCAS-85) benchmark suite. The circuits in ISCAS-85 deliberately provide confidentiality and abstraction of high-level structural design or random circuit logic (Hansen and others, 1999:72). We use combinational circuits as an abstraction of random programs because they are deterministic in nature and provide visually understandable intermediate structural information. Structural randomness is achieved by using a seeded random number generator that selects two random node points within the circuit and connects them using a randomly selected two-input gate. We use six gates, known as the circuit's basis, to construct our random circuits: AND, OR, XOR, NAND, NOR, and NXOR, each with different I/O characteristics. It is possible to specify the number of input bits and the number of output bits in the random program generator (RPG) so we can compare the I/O characteristics to non-random programs of equal size.

Analysis of the black-box characteristics are selected metrics from the NIST suite of metrics to evaluate pseudo-random number generators. We examine each bit of the output as a random string output. The rationale behind this decision is that if it is possible to correctly guess every output bit, then it is possible to correctly guess the output. We selected metrics that did not require minimum bit string lengths. Because the output sequence affects some metrics and the output sequence is directly dependent on the input pattern sequence, we use two input different sequences for each set of input. First, we used a standard binary counter in big-endian order. We then converted this pattern into gray-code as a second input pattern so we could observe the avalanche or diffusion effect of a single input bit.

This research then examines the applicability, efficiency and security of CFT as a software-only software protection method. First, we explore the applicability by replacing Java methods with function table implementations. Efficiency is a qualitative measurement of the performance of the replacement and the memory space ratio compared to the original. Finally, security is quantitatively measured by evaluating the randomness of the output in the obfuscated version compared the un-obfuscated version. In addition, we compare decompiled functions of the original program against decompiled functions of the obfuscated versions using three open-sourced Java decompilers in a qualitative analysis of security.

Our hypothesis is that random programs generate random output. Therefore, the intent protection model is necessary because structural randomness is insufficient. Furthermore, CFTs using symmetric key encryption tables can be effective at white-boxing and black-boxing bounded input-size functions. Each implementation, however, will be computationally expensive to generate because full enumeration of the function input space is necessary. A mitigating factor is that we only need to compute encryption function tables once and we can reuse the encryption tables to protect different functions.

3.9 System Boundaries

Since the goal of this research is to examine obfuscation under the intent protection model, the obvious system boundary consists of the components needed to create and evaluate obfuscated functions. This system, the Encrypted Program Generation Engine (EPGE) has two parts because intent protection models obfuscation as two parts, white-box protection and black-box protection.

The first part of system generates random circuits to emulate random program structure using the aforementioned six gate circuit basis. We verify uniqueness of the generated programs using a CRC32 hash. For clarification, we consider symmetrical circuits unique using this method. For example, an AND gate connecting node one and node two is considered distinct from an AND gate connecting node two and node one even though they would generate the same results. This was a design decision made so that uniqueness of a circuit could be quickly determined using the hash checker. For this research, the RPG used is currently implemented in C++ which generates BENCH circuit files that are then interpreted and translated into a Java object for analysis by the EPGE. The EPGE then runs a black-box attack using the binary counter sequence and the gray-coded binary counter sequence. Results from the two operations are analyzed using seven statistical measures adapted from the NIST pseudo-random number generator test. We then perform the same set of analysis on any deterministic function of bounded input size implemented within the EPGE to obtain black-box characteristics of the function.

The second part of the system constructs CFT obfuscated versions of functions implemented within the EPGE. Ideally, the EPGE should be able to read a Java class file that contains the high-level source code of a deterministic function and build a corresponding CFT version in high-level source for replacement in the original. Due to time constraints for this research however, a deterministic function must be built within the EPGE package. Since the goal in this part is to quantitatively examine function algorithms through a Java decompiler, manually importing algorithms in the EPGE should still adequately provide the observable results for white-box security comparison. For encryption, we use an open-sourced implementation of the AES algorithm and verify its functionality using the KeySBox Know Answer Test Values (Bassham, 2002). In addition, the EPGE is capable of generating BES equivalent of a function table using the Quine-McCluskey algorithm. We select the Quine-McCluskey algorithm because we know the algorithm's complexity ("Quine," 2007). While we do not use it directly as a security mechanism, it is a component can be used to generate an obfuscation implementation.

Thus, the main component under test is the EPGE since it is the component that generates the random programs, generates CFT obfuscated programs, and performs black-box and white-box analysis. Additional component consists of the Java compiler, the Java virtual machine, and hardware, such as the CPU and memory running the virtual machine.

We note that the source code of the pre-obfuscated function is an input to the EPGE and therefore not considered as part of the system under test (SUT). The EPGE could obfuscate a function's function table if it was available which means that the source of the function does not influence design of the engine. We illustrate the complete SUT in Figure 14.

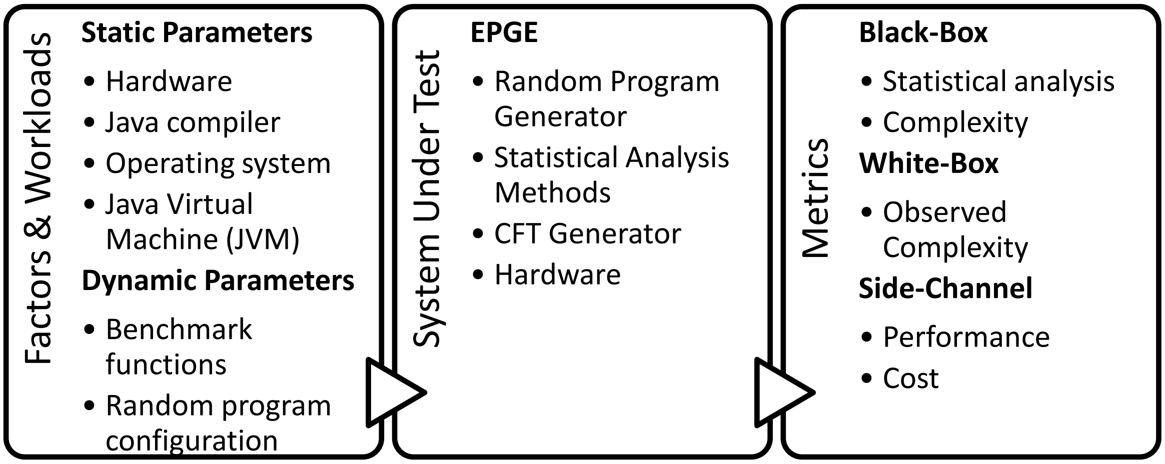


Figure 14. System Under Test

The main block contains the EPGE, the Java compiler and JVM, and the hardware components. We show the changing factors and workloads as inputs on the left while the outputs are the metrics for the SUT which appear on the right of the main SUT block.

3.9.1 System Services

The EPGE takes a source-level function and generates a set of random programs with equal input/output size. We perform statistical analysis on the output of the original function and the output of the random program set. We list in Table 5 the possible outcomes.

Table 5. Possible Outcomes for System Services

Result	Random Program Set	Original Function
1	Statistically random output bits	Statistically random output bits
2	Statistically random output bits	Statistically non-random output bits
3	Statistically non-random output bits	Statistically random output bits
4	Statistically non-random output bits	Statistically non-random output bits

We select an AES key and compose it with the output of the original function to produce a CFT. We then perform statistical analysis on the CFT.

3.10 Workload and Factors

As stated in the problem definition, a benchmark suite of programs does not yet exist for software obfuscation. Previous and concurrent work performed by the Program Encryption Group (PEG) uses the ISCAS-85 circuit library in BENCH format. For continuity and compatibility with ongoing experiments, this research includes a function that reads in a bench circuit file and reproduces its input and output pattern. It also works in tandem with the RPG to construct random structure in BENCH format using two-input gates.

Because most ISACAS 85 circuits are too large to enumerate within the resources available to this research, additional benchmark programs of deterministic functions are proposed. Simple equations, such as *y = a * b + c*, trigonometric functions, such as *y = cos(x)*, and the Fibonacci sequence are considered because they can be bounded in input-size and were used in related research works focused on confidentiality of execution (Christiansen and others, 2006:2; Torri and others, 2007; "Lookup," 2007). Table 6 summarizes the workload factors as well as the reason for their selection as benchmark functions.

Table 6. Benchmark Functions

Program	Reason for selection as benchmarks
y = a * b + c	• integer computation • simple arithmetic • divisible into f_1 and f_2
y = cos(x)	• periodic output pattern • floating point arithmetic • historically represented as table lookups • often used in digital processing algorithms
Fibonacci	• recursive or iterative function • expands quickly
ISCAS 85 circuits	• hardware analogy

3.11 Metrics

3.11.1 Black-box Metrics.

Black-box metrics were adapted from a NIST test suite for pseudo-random number generators. For clarification, we consider each output bit as a generator of a bit string and use statistical analysis to determine if patterns exist for each bit enabling the adversary to guess subsequent bits within the bit string.

We list in Table 7 the statistical tests used in this research and a summarized explanation for each test ("National Institute," 2007). We recognize there are existing test suites such as JDieHard, NESSIE, and the one provided by the NIST; however, these suites were designed for random program generators and some tests required minimum bit string lengths of 10,000 bits or greater. Thus, we had to selectively implement tests that could provide results on much smaller bit string lengths due to our experimental benchmarks that have a relatively smaller input space.

Table 7. Statistical Test to Analyze Function Output

Test	Explanation
Frequency (Sequence)	Ratio of 1's to 0's produced in an output bit
Frequency (Output)	Ratio of 1's to 0's produced by all output bits
Longest runs of 1's	Longest uninterrupted sequence of 1's
Number of 1's runs	Number of runs with uninterrupted sequence of 1's
Maximum excursion	Greatest distance from zero achieved when each output resulting in 0 or 1 is mapped to -1 and 1 respectively and the output's bit string is summed.
Excursion states	Size of the set of distances from zero achieved when each output resulting in 0 or 1 is mapped to -1 and 1 respectively and the output's bit string is summed.
Zero excursion cycles	Number of zero excursion cycles. A cycle the summation of the outputs to an *m*-th bit and back to the origin when each output bit resulting in 0 or 1 is mapped to -1 and 1 respectively; *m* is increased incrementally until it reaches the end of the bit string.
Approximate entropy	Percentage of output bits flipped when a single input bit is flipped; used when gray-code input is used.

3.11.2 White-box Metrics.

Metrics to evaluate the randomness of a structure is the subject on concurrent research within the PEG research group. Because the proposed approach removes the program structure by converting a function into a two dimensional representation, this research can subjectively examine the structural obfuscation of using CFT using various Java decompilers. We derive quantitative security measures from the steps that an adversary needs to perform to break apart the CFT along with the computational complexity associated with each step, as stated in section 3.5; the theoretical maximum security according to computational complexity is directly correlated to input size.

3.11.3 Side-channel Metrics.

Performance and memory costs are important because they determine the practicality of the obfuscation. We measure performance as execution time in seconds and measure cost in terms of memory size in bits. These metrics are common, non-subjective, and understandable within the computer science. Because cost of the CFT implementation is very different from the time it takes to *generate* a CFT implementation, a developer must decide whether generation costs should factor into the cost of obfuscation. For consistency, we only consider the memory cost of the deployed obfuscation and the performance running the obfuscated function when evaluating an obfuscated program. We note that multiple obfuscations of different functions will cost less to generate because the paired encryption table only needs to be generated once. Thus, future obfuscations of functions with the same bounded input-size incur incrementally less generation costs because we can pair it with any pre-enumerated encryption table.

We compare the above metrics against the four properties of an obfuscation proposed by Collberg and others' work summarized in Table 8 (Collberg and others, 2004:738).

Table 8. Summary of Collberg and Others' Obfuscation Properties

Properties	Explanation
Potency	Difficulty in understanding the obfuscated code
Resilience	Difficulty in automating a tool to de-obfuscate obfuscated code
Cost	Penalty in execution time/memory space incurred by obfuscated code
Stealth	Statistical similarity of obfuscated code compared to pre-obfuscated code
Quality = (Potency, Resilience, Cost, Stealth)	

In using CFT, potency and resilience should be a direct correlation to the time required for an adversary to conduct a brute-force attack on the function table of *n*-size input disassociating security metrics from cognitive ability. We do not emphasize stealth in our approach because security should not be dependent on obscurity ("Kerckhoff's principle," 2007), even in the watermarking configuration. We postulate that CFT's stealth is directly related to the number of lookup operations that are done within non-sensitive portions of the code because CFT operations are lookup operations; the lookup operations may be file accesses or array searches depending on the final implementation of the CFT.

3.12 Parameters

Because the EPGE provides metrics for the CFT obfuscated function in relation to the original function, the hardware that executes both functions must remain constant across all experiments. The experimental system is an Intel M 1.73 GHz processor, with 1.5 gigabytes of RAM running Windows XP Tablet Edition. The operating system is important because it is not possible to control task scheduling which affects performance metrics; thus, we repeat experiments and average results to minimize the variability caused by the operating system. We use the Java 5.0 compiler; this is also important because we perform analysis on the bytecode code produced by the compiler.

3.13 Evaluation Technique

We divide security achieved through CFT into white-box and black-box security. CFT obfuscated functions are decompiled with Jadeclipse, JDec, and JODE to compare the original function against the obfuscated function. This step provides a subjective measure of white-box security or how well CFT "confuses" someone using standard packages of these readily available open source tools. We select three open source Java decompliers; though their main function is decompilation, each has distinguishing features. Jadeclipse[1] is available as a packaged plug-in for the Eclipse Java development environment. JODE[2] is available both as an applet and console application. JDec[3] has an easy-to-use graphical user interface. The first and most notable Java decompiler, Mocha[4], was not used because it has not been updated to decompile recent changes in Java class files.

We assess black-box security by the same statistical random measures used in evaluating the output of random programs. It is difficult to otherwise evaluate security efficacy other than this manner because CFT abstracts all other implementation detail specifically to force the adversary to do computationally expensive brute-force attacks. Using a "red-team" of skilled reverse-engineers would provide much better evaluation of

[1] http://jadclipse.sourceforge.net/
[2] http://jode.sourceforge.net/
[3] http://jdec.sourceforge.net/
[4] http://www.brouhaha.com/~eric/software/mocha/

the CFT approach; this evaluation approach is unfeasible due to the limited time and resources available for this research.

We use direct measurement of execution time because it is the most understandable metric for computer programs; we collect this measurement using the Eclipse Test and Performance Tool Platform (TPTP) profiling tool[5]. Finer measurements, such as calculating instructions per clock cycle, are difficult due to non-standard processor instruction sets, pipelining, caching and operating system tasking order. In addition, computer scientists and cryptographers generally use theoretical complexity metrics to describe algorithms in a hardware platform independent notation.

File size is also a well understood metric and we use the standard file explorer on the Microsoft Windows XP installed on a NTFS file-system partition.

3.14 Experimental Design

The EPGE performs the experimental design on the benchmark programs. In the steps outlined in the approach section, we compare the output of the original function the RPG set of functions with equal input/output size for black-box analysis. Since most of the benchmark functions are not actually BENCH circuits, we used a constant intermediate size as a parameter in the RPG. For each random program set, we generate 1,000 unique circuits to provide output for comparison by statistical analysis. We then perform statistical analysis on AES ciphers with 1000 different keys to verify the effect of encryption on functional output bits.

[5] http://www.eclipse.org/tptp/

3.15 Chapter Summary

This research examines intent protection as an alternate obfuscation model to VBB deemed ideal but impractical for practical obfuscators. Intent protection makes a clear distinction between a program's white-box structure and black-box functionality; this makes it possible to evaluate these two facets of a program separately. Using pseudo-random number generators and encryption as an analogy to obfuscation, this research applies security metrics independent of human cognitive ability. We propose function tables and compositions of function tables with symmetric key encryption tables as an obfuscation approach to completely mask white-box structure. Encryption algorithms provide resistance to linear and differential analysis on the output of the function. We keep the hardware, operating system, and compliers constant in the SUT to reduce variability in factors generating the metrics. After collecting the metrics, we can use the data to justify further development of this approach. This chapter presents an alternate model for evaluating obfuscation, the creation of the CFT implementation, the security principles for its design, and the experiments to examine CFT as a software-only software protection technique.

IV. Analysis and Results

4.1 Chapter Overview

This chapter discusses the results of the experiments described in Chapter III. We use statistical data analysis to measure the output generated by randomly generated bench circuit functions, benchmark functions, and functions composed with an AES encryption. Additionally, we use three open-sourced Java decompilers to observe the effect of CFTs on the Java class files from compiled benchmark functions. Our observations qualitatively measure the CFT's protection strength and propose improvements to this approach.

4.2 Results of Experimental Benchmark Programs

4.2.1 Quantitative Analysis of Black-box Data.

We analyze the sets of 1,000 randomly generated BENCH circuits with the statistical tests listed in Table 7. We use the input/output sizes of the benchmark programs listed in Table 6 as parameters for the RPG. Because the impact of the internal structure is currently unknown, we set the parameter for the number of intermediate nodes to 100, 300 and 500. For the circuit c17, we generated an additional random program set with six intermediate nodes to match the original circuit description.

First, we conduct an analysis on the collective random function output. Each function produces an output signature which is the output sequence of the function based on an input sequence. The total possible number of unique signatures is $(2^{out})^{(2^{in})}$

where *out* is the number of output bits and *in* is the number of input bits. We check the output signatures of the random function sets for uniqueness using a CRC32 checksum.

Numbers within sets of identical output signature are an indicator of functional equivalency and structural diversity. For a set of randomly generated programs to produce large sets of non-unique output signatures, it may be a signature that exhibits weak correlation between structural pattern and output signature. If we intend to obfuscation white-box information by emulating randomly constructed circuits, then signatures with a large number of candidate structures are good candidates for obfuscation. In practical terms, it means that we can swap the structure of one member within the set with another member in the same set. This obfuscates the original structure because we produced the alternate structural logic randomly without any knowledge of the original structure and therefore the replacement structure cannot leak information about the original structure.

Random function sets of 5-2-6 and 5-2-100 yielded 125 and 71 functions that produced non-unique output signatures respectively presented in Table 9. The other random function sets did not produce any duplicate output signatures.

Table 9. Non-unique Output Signature Characteristics of 1000 Random Functions

5-2-6		5-2-100	
Set size of identical output signatures	**Number of Sets**	**Set size of identical output signatures**	**Number of Sets**
2	**32**	**2**	**7**
3	**2**	**11**	**1**
4	**8**	**12**	**1**
5	**1**	**16**	**1**
9	**2**	**18**	**1**

We expected fully unique signatures for even the small input and output parameters because 4^{32} unique signatures are possible. For a set of 1,000 random functions to exhibit signature collisions may indicate that structural diversity is great for smaller input/output parameters. We observe that the intermediate node is a factor in producing signature collisions. Increasing intermediate node size causes a drop in collision frequency but an increase in collision concentration where the chance of collision is less likely, but in the case of collision, the collision set tends to be greater in size. We graph our observations regarding intermediate node size and signature collisions in Therefore, obfuscation of a complete white-box structure may be more effective with partial obfuscations of smaller input/output size with a large intermediate node size so there are several candidates for replacement.

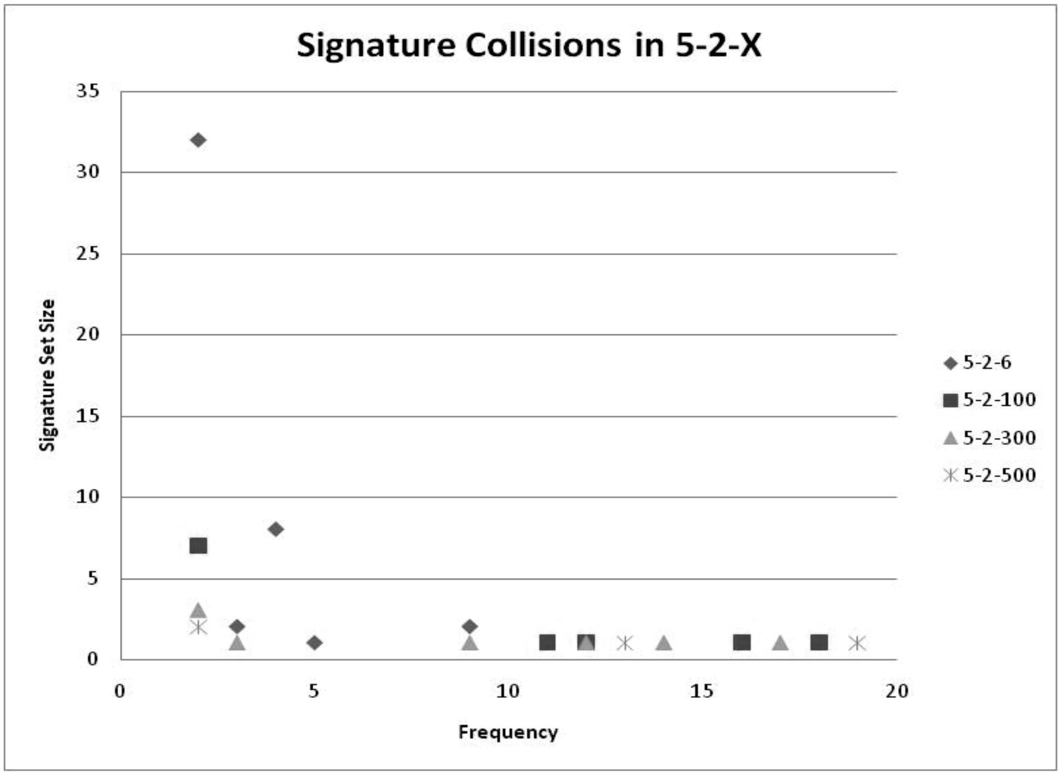


Figure 15. Signature Collisions in 5-2-X

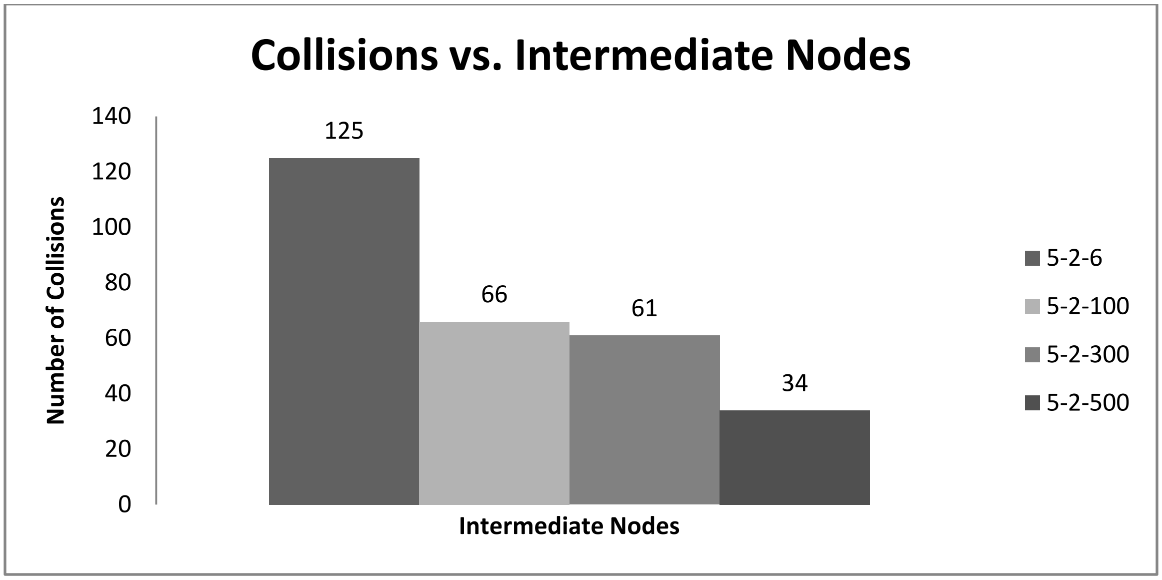


Figure 16. Signature Collision to Intermediate Node Size

We then perform analysis on each individual benchmark program and corresponding function set. For clarity, we display only the gray code input sequence in the following tables. It is important to note that metrics on run lengths and excursion states are dependent on input sequence. In addition, using gray code input provides the avalanche metric for comparison between the benchmark output and random function output. We recognize that there are many there are many possible sequences that exist where we flip only one input bit. We use the gray code as an exploratory technique to observe the avalanche affect of input bits; the avalanche effect on output bits for cryptographic ciphers should be observable using gray code input. We verify by using a black-box analysis of the output from 1,000 AES encryption output tables using a gray code input sequence. Tables illustrating the results of the statistical analysis comparing benchmark and respective input/output size random functions are found in the Appendix; the result of each test by output bit is provided so that the distinction between benchmark and random functions can be visualized. We use averaging across the 1,000 random functions on each output bit to provide a result. The experiments provide a picture of the expected values of the seven statistical tests for a randomly generated program of a certain input/output size. From the results of this experiment, it appears that random functions generate consistent results for each output bit across all tests which can be contrasted against the output bit behavior in the benchmark functions.

In Figure 17, Figure 18, and Figure 19, we graph the standard deviation for all bits in the output by test for some of the benchmark functions and their respective random program set. For these graphs, we included the binary counter sequence. We observe disparity in results; random program sets produce significantly less diversity in their output bits than the benchmarks as shown by the flat lines generated by the random program sets in the three figures. We note that our two input sequences produced similar results.

Within this limited set of benchmarks, it appears that the number of excursion states is the biggest indicator of an unprotected benchmark function versus the set of random functions while the number of zero cycles tends to be a poor indicator. In addition, this black-box analysis on deviation from expected randomness values lets us know which statistical test best isolates non-random behaving bits in the output. We can then target the control flow of the bits that do not exhibit random behavior with structural randomness. This information is useful in cases where we cannot use black-box protection and the security must rely only on white-box structural entropy.

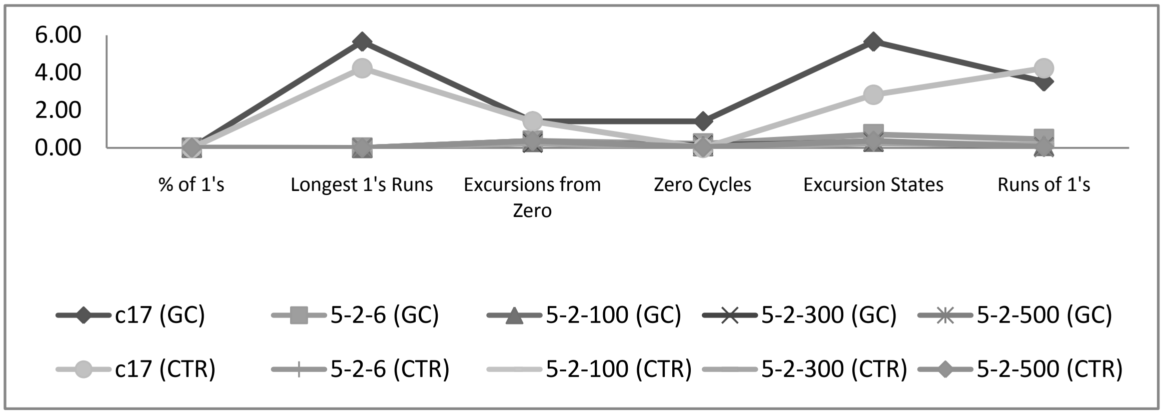


Figure 17. Standard Deviations of All C17 Output Bits by Metric

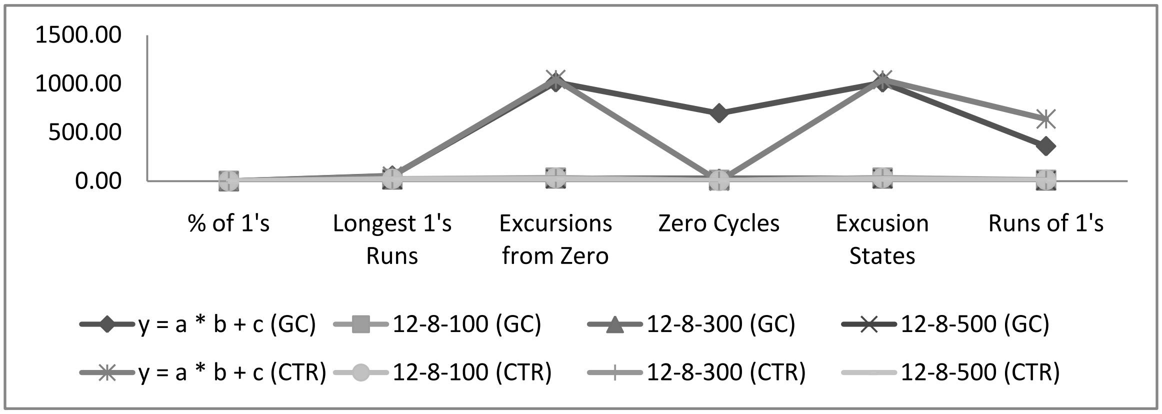


Figure 18. Standard Deviations of All *y = a * b + c* Output Bits by Metric

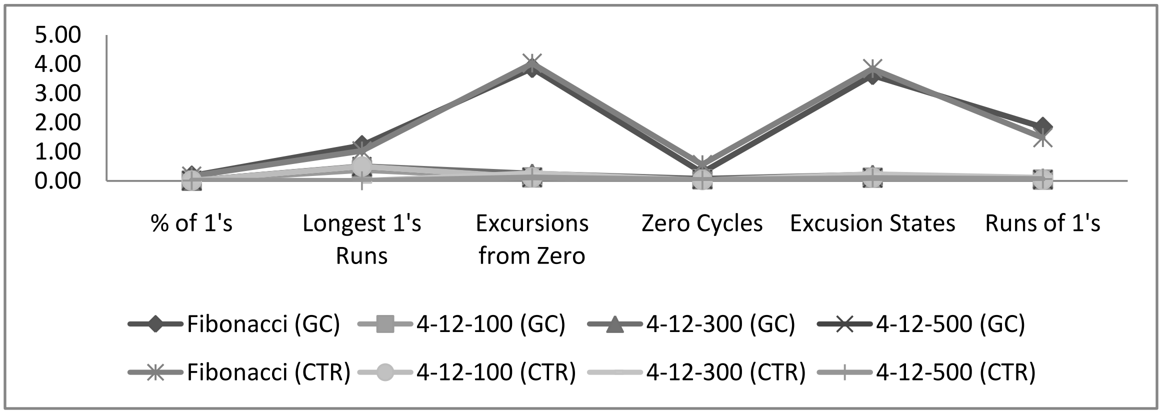


Figure 19. Standard Deviations of All Fibonacci Output Bits by Metric

We conducted a statistical analysis of AES encryption with 1,000 keys and equal input size of five bits to examine the feasibility of protecting a c17 circuit from the ISCAS-85 circuit library with AES. The standard deviations between AES and the random program set for each metric, shown in Figure 20. Standard Deviations of All AES Output Bits by Metric was significantly closer to zero than any other experimental function.

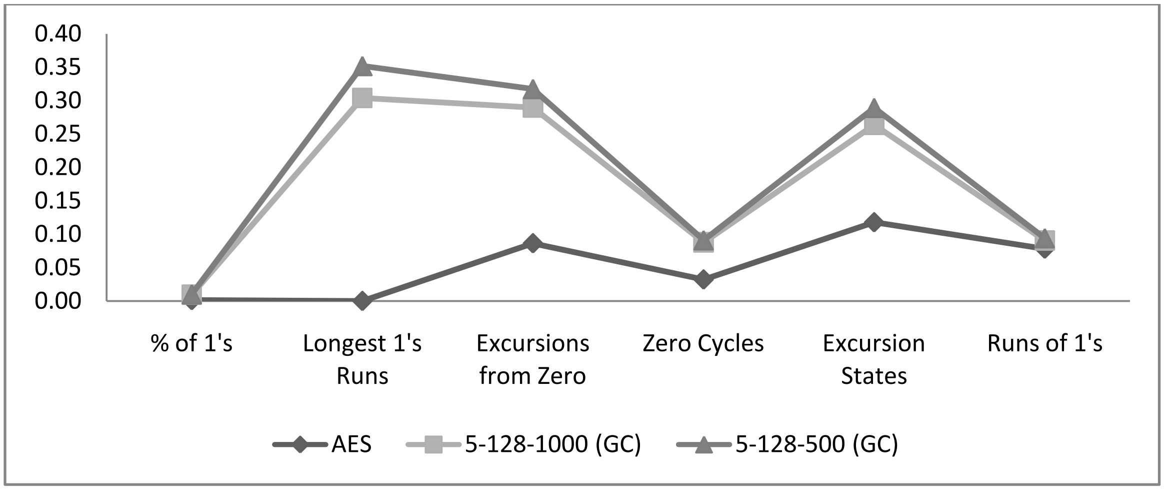


Figure 20. Standard Deviations of All AES Output Bits by Metric

The averages and standard deviations can also be found in Table 10; per bit graphs are not included because it is difficult to clearly represent all 128-bits graphically. We note that we adjusted the random program set parameter from 100 and 300 to 500 and 1000 in order to accommodate the significantly larger output size in AES. Different results between the AES and random program set produce approximately the same results. The metrics provided by these random sets are valuable because the results for these metrics are unknown for random program structure. Thus, these metrics provide a comparison point for functions that may have the parameters such as input size, output size, or intermediary node size.

Table 10. Statistical Results of AES and a Random Program Set

Function	% of 1's	Longest 1's Runs	Excur. from Zero	Zero Cycles	Excur. States	Runs of 1's
AES avg	0.50	4.00	6.62	0.69	8.96	8.25
AES std dev	0.00	0.00	0.09	0.03	0.12	0.08
5-128-500 avg	0.50	8.05	18.34	1.23	19.51	4.55
5-128-500 std dev	0.01	0.35	0.32	0.09	0.29	0.09
5-128-1000 avg	0.50	7.95	18.35	1.12	19.51	4.60
5-128-1000 std dev	0.01	0.30	0.29	0.09	0.26	0.09

We note that the metrics did not change significantly between the 500 and 1000 internal node set or random functions indicating that intermediate node size may not be a significant factor on the randomness of individual output bits. This was also true for the benchmark programs even though we did produce a small percentage of signature collisions in the 5-2-X set of experiments. Standard deviations also remained small though we note that the standard deviations of the two random set in our 5-2-X with AES experiments mirrored each other which could indicate that our RPG construction is a factor. No functions within the two 5-128-X sets shared the same output signature.

In addition, the test verified that the 1,000 AES keys produced 50% approximate entropy on the output as expected when we use gray code input. We note that the unprotected benchmark functions on average produce only 26% approximate entropy. This means that a change in a single input bit has significantly less impact, or more specifically, less of an avalanche effect on the output bits of randomly generated circuits. Therefore, our results indicate that structural entropy alone does not, on average, produce the same black-box entropy as cryptographic functions. We are interested in the approximate entropy results specifically because the greater entropy tends to hinder black-box analysis. We graph our results in Figure 21. The first column is our verification of approximate entropy on AES, followed by the approximate entropy observed in our randomly generated sets. We obtained the fourth column results by using an AES encryption table to protect the output of the c17 circuit. This did not increase approximate entropy because ECB does not hide output patterns. We achieved approximate entropy results similar to AES when we applied two different padding schemes to diffuse the output space prior to applying the AES encryption, as shown in the last two columns.

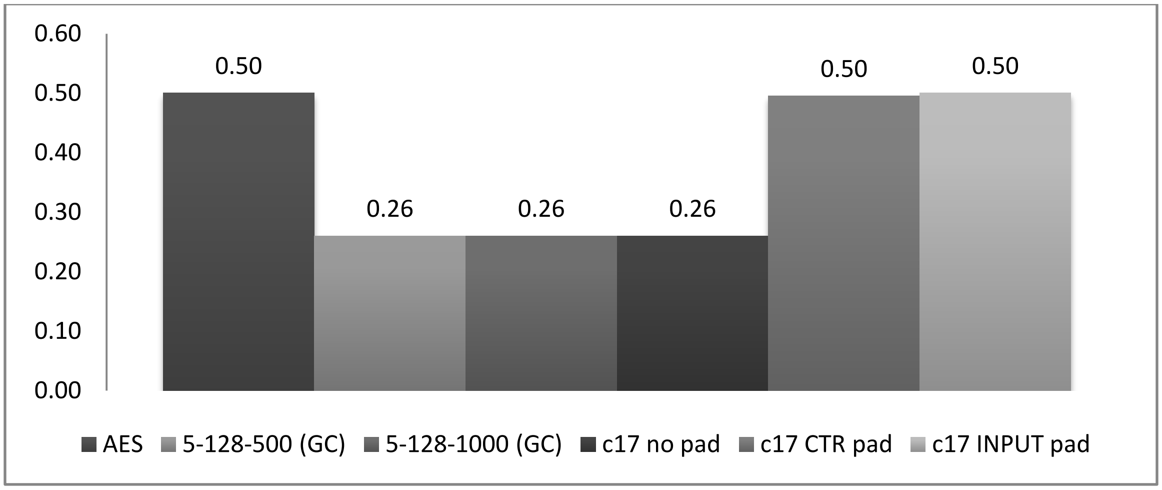
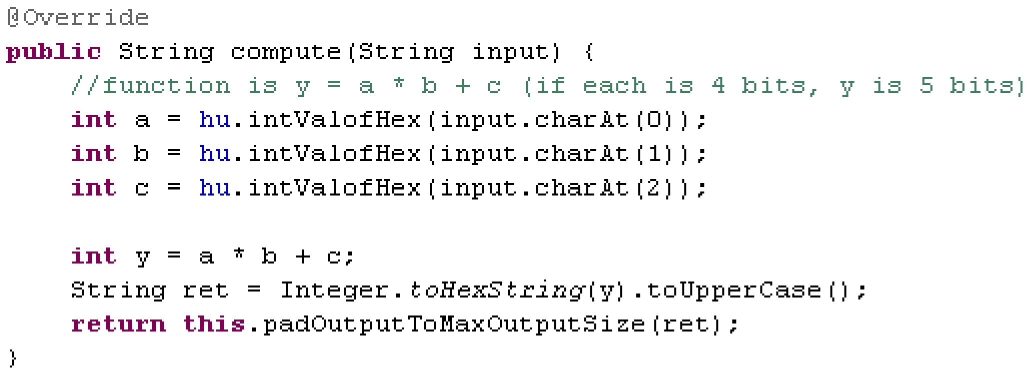


Figure 21. Approximate Entropy of AES and 5-128-X

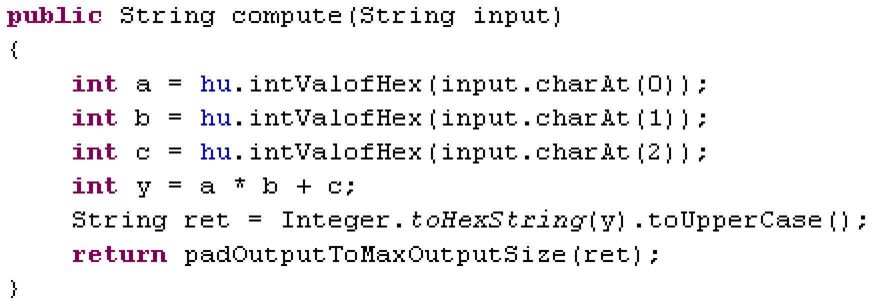
4.2.2 Qualitative Analysis of White-box Data.

None of the Java decompilers had any difficulty creating source code from unprotected and CFT protected code. We expected this because CFT was a technique used to remove the structural information rather breaking the tools' decompilation process. The common denominator between the original source and the results provided by all the decompilers is the removal of comments. Figure 22 shows the original unobfuscated Java source code for one of the benchmark functions for contrast with Figure 23. JODE and JDEC provided the same results. Figure 24 is the decompiled source by Jadclipse on a CFT implemented c17 benchmark function. The CFT code is simple in software engineering complexity and the function becomes a file access in this particular implementation.



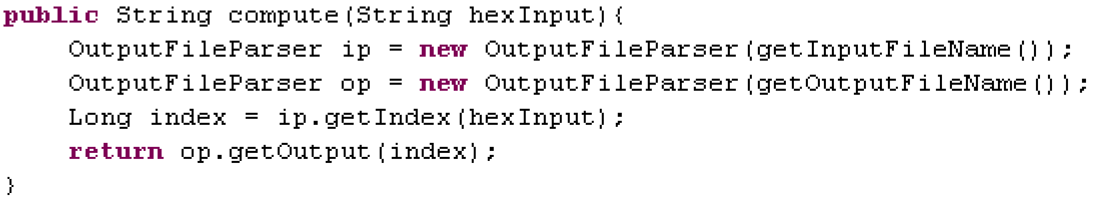
```
@Override
public String compute(String input) {
    //function is y = a * b + c (if each is 4 bits, y is 5 bits)
    int a = hu.intValofHex(input.charAt(0));
    int b = hu.intValofHex(input.charAt(1));
    int c = hu.intValofHex(input.charAt(2));

    int y = a * b + c;
    String ret = Integer.toHexString(y).toUpperCase();
    return this.padOutputToMaxOutputSize(ret);
}
```

Figure 22. Original Source Code of *y* = *a* * *b* + *c*



```
public String compute(String input)
{
    int a = hu.intValofHex(input.charAt(0));
    int b = hu.intValofHex(input.charAt(1));
    int c = hu.intValofHex(input.charAt(2));
    int y = a * b + c;
    String ret = Integer.toHexString(y).toUpperCase();
    return padOutputToMaxOutputSize(ret);
}
```

Figure 23. Decompiled Source Code by Jadclipse



```
public String compute(String hexInput){
     OutputFileParser ip = new OutputFileParser(getInputFileName());
     OutputFileParser op = new OutputFileParser(getOutputFileName());
     Long index = ip.getIndex(hexInput);
     return op.getOutput(index);
}
```

Figure 24. Decompiled Source Code of CFT implementation

4.2.3 Analysis of Side-Channel Data.

CFT is not fixed to an implementation because the security concept is to prevent adversary analysis by flattening of functional structure to two-dimensions. Because this research implemented the CFT using text files, the protected programs took longer to run due to frequent file accesses; the disk accesses incurred cost penalty in performance time because disk access operations are slower than the operations in the benchmark programs which did not require significant processing power.

In complexity terms, a lookup operation in the encryption table is constant time, O(1) making CFT very scalable. Constant time is achievable because every entry is the same size and we can provide the entries, input order sorted, so that an index search is possible. We can use the original cryptographic primitive to decrypt and recover the output and we know that the cryptographic primitive runs in polynomial time. If we use the function table for decryption, we could first apply sorting to the output table and then use a binary search to achieve O(*n*log*n*) performance. We cannot use the same indexing method as the encryption table because the ciphertexts sparsely populate too large a range.

We found the file sizes consistent to our estimates of $2^n * m$ bits where n is the number of input bits and m is the number of output bits. We note that a side effect in our implementation under the NTFS file system test environment is that Windows file explorer reports a difference between the actual file size and the size the file takes on the disk.

For BES representations of CFTs, we found early in our experiments that storing the BES as a file take much more memory space than the CFT in our implementation. For a BES, we cannot estimate the length or the number of prime implicate for each output bit. However, we are attempting to achieve random output so we expect each output bit to produce significantly long Boolean equations making textual representation very inefficient. We do not propose BES implementation as a text file; we generate it as a blueprint for a minimized sum-of-products two-dimensional gate structure that can be then implemented as code. We implement BES textually mainly to examine this structure generation for future experimentation. In terms of performance, BES runs with complexity O(*n*) where *n* is the number of output bits because each output bit has its own Boolean equation that runs in constant time.

4.3 Summary

The research shows that random programs can be a comparison tool for intent protected obfuscation techniques such as CFT. While there is yet to be a set of agreed upon metrics to compare program structure, there are metrics in use that analyzes function output. The results shown in this chapter show that programs with randomly generated structure produce randomness across the output bits. The randomness closely equals that of AES, a strong encryption algorithm. In the same way that functional randomness produces output that is hard to discern a pattern, structural randomness may produce program structure that is difficult to analyze. Thus, if it becomes possible to accurately assess structural randomness, it will be possible for an obfuscation to be intent protected by creating an obfuscated version of a function that is both structurally random and functionally random. In the absence of such metrics, this research uses CFT with symmetric encryption to remove the structural details of a program while creating measurably random output as an obfuscation technique.

V. Conclusions and Recommendations

5.1 Chapter Overview

This chapter reviews the main research goals outlined in Chapter I with the corresponding findings of this research. For each goal, we briefly summarize the relevant results and conclusions. We propose recommendations for future research and enhancements as well.

5.2 Research Goals

5.2.1 Describe an Alternate Model for Software Obfuscation.

This research describes an alternate model for obfuscation and its possible applicability. Its authors and the theoretical obfuscation community, in general, accept the original VBB model, to be a non-pragmatic model for building obfuscation tools. The intent protection model proposes removal of the function preservation property as a modification to the VBB model. This research investigates the problem where black-box information may provide reverse engineers information to reconstruct a protected function's white-box information suggesting that masking a function's input and output relationship is critical to protecting the function.

5.2.2 Describe an Implementable Obfuscation Algorithm.

Every deterministic function generates a function table which describes the function in a two-level representation that removes all white-box information. We select the AES symmetric key cryptographic primitive to black-box protect deterministic functions due to their strength to key length ratio and understandability when operating in

ECB. We provide motivation where the applicability of function tables are within the confines of bounded input size and deterministic algorithms.

5.2.3 Quantitatively Assess Obfuscation Quality with Non-cognition Metrics.

By dropping the function preserving property from the VBB model in the intent protection model, we can use cryptographic metrics, such various statistical analyses of function outputs, in lieu of cognition and psychology-based metrics. Comparisons between original functions and randomized functions of the same characteristics show that the black-box information can leak information about the program without white-box analysis. Random program set and output bits of the AES algorithm both produce output bits that are statistically random; a CFT that composes the function with an AES function table can emulate similar output results in addition to removal of structural information thereby giving the adversary as limited information as possible for analysis. CFT is also understandable and low in software engineering complexity.

5.2.4 Qualitatively Assess Obfuscation Quality with Reverse-engineering Tools.

The function table and composite function table approach is very effective against Java decompilers such as Jadclipse, JODE, and JDEC. As expected, a table representation flattens the function into a LUT, leaking no white-box information other than the fact that we used an LUT in the software-only environment despite using an information laden bytecode language. Function table also flattens the run time characteristics thereby reducing the side-channel information that may inadvertently leak details about white-box structure.

5.3 Conclusions of Research

Software obfuscation is a difficult problem and we concede that a single perfect solution to secure all programs does not exist. An agreed upon set of metrics and benchmarks are needed to evaluate various proposed techniques in software protection research.

This research advocates the use of the intent protection model to evaluate software obfuscation in place of the idealistic VBB model. Under intent protection, it is possible to use separate but established black-box metrics to accurately define security strength of obfuscation so that we can achieve practical obfuscation with provable security measures. Applying symmetric cryptographic principles to obfuscation reduces the dependency on using software engineering metrics that were not intended for use as a security metric. Using function tables is a technique that heavily favors security and performance over applicability in the obfuscation trade-off of applicability, efficiency, and security. By converting all deterministic functions to a two-dimensional representation, we protect the white-box information from analysis. We note that we do gain some applicability in that it is easier to determine a function's input/output size than it is to find if the function is part of a family that can be obfuscated in other theoretical models. We also note that the CFT approach benefits developers, such as nation states, who can leverage their asymmetrically vaster computing resources to create protected functions of larger sizes or to break the protection under the CFT technique more quickly.

5.4 Significance of Research

The limitation of the functional table approach is the treatment of all programs as deterministic functions and the severe bound on input sizes. It is noted however, that this technique was intended to serve as a demonstration platform for the intent protection model and identify the tools to evaluate both black-box and white-box security for Java programs. This research directly supports PEG's obfuscation research by first identifying the tools to evaluate output entropy and then designing the benchmarks programs for testing in the software domain. Furthermore, this research has demonstrated metrics to functionally evaluate random programs which benefit concurrent PEG research in evaluating the structure of random programs.

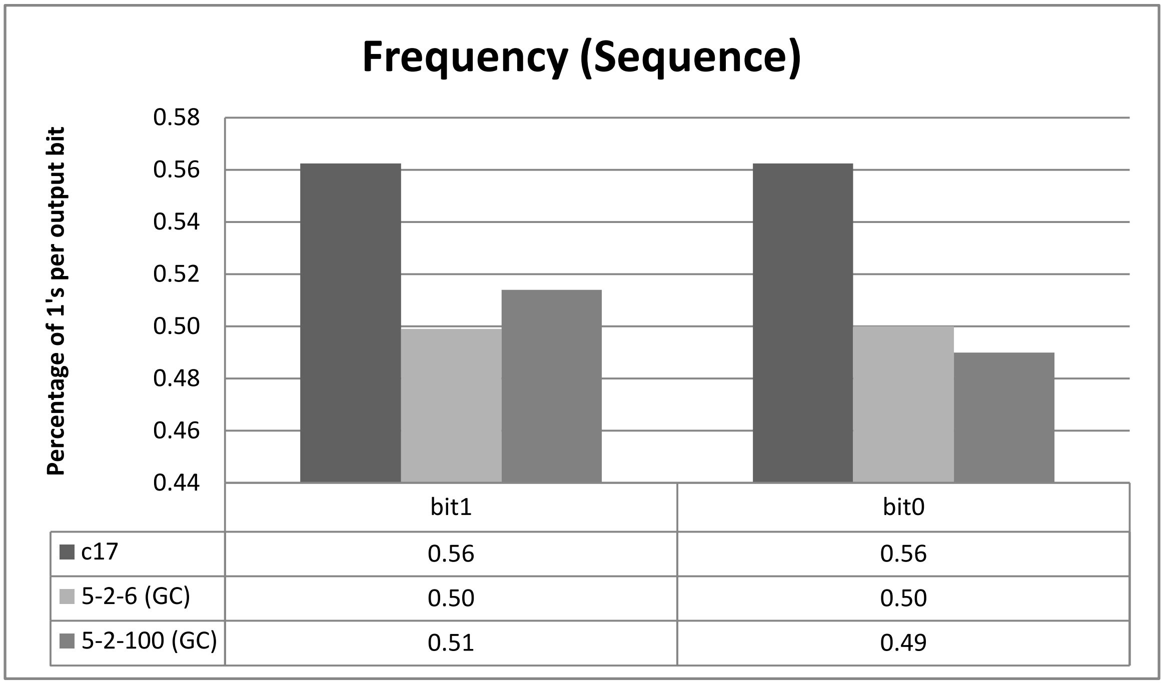
5.5 Recommendations for Future Research

Our intent was to use functional tables as a software-only solution for a software problem. It is foreseeable that this approach would be adaptable in the hardware domain due to the prevalence of FPGAs and their inherent LUT structure where it is possible to replicate an *n* input size combinational circuit with an *n* address-input memory (Valhid, 2007:106). In addition, because hardware can concurrently compute at the bit level, the same computation can be executed hundreds or thousands of times faster in comparison to a microprocessor. However, it is noted that white-box information such as hardware characteristics not present in the software environment may be leaked through the functional table implementation and thus deserve further investigation.

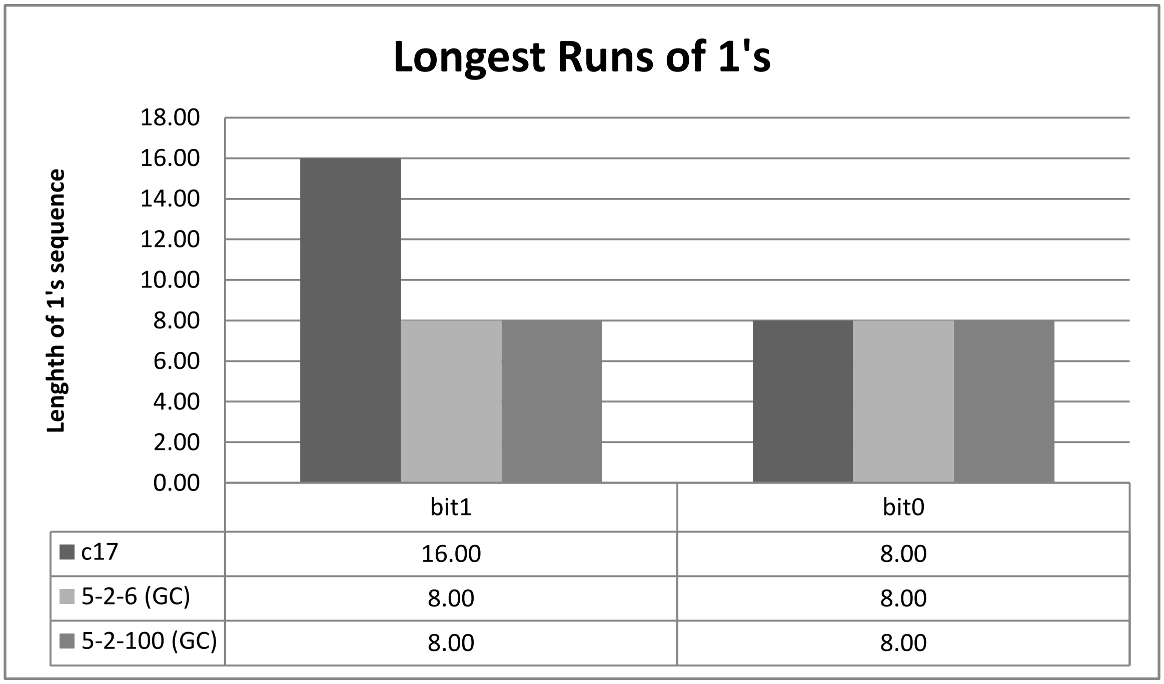
5.6 Summary

This research shows that intent protection model is a viable alternative to the VBB obfuscation model. The proposed function table approach is a provably secure technique that we can evaluate with established cryptographic metrics. It is also understandable in approach and implementation. While the approach imposes restrictions on the applicability to certain programs, function tables serve as a foundation to bridge the theoretical obfuscation research and the practical obfuscators. A complete obfuscation based on both output entropy and structural entropy may provide a multi-tiered defense against reverse engineer targeting sensitive military software.

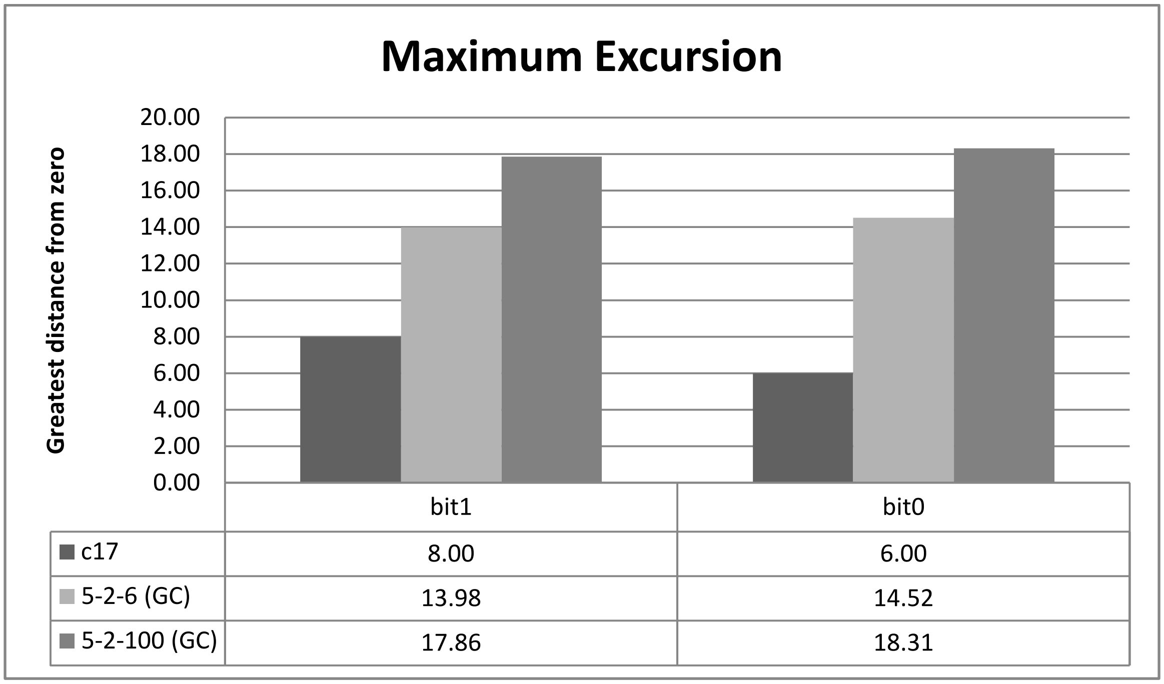
Appendix A: Black-box Analysis of c17 Against Random Functions



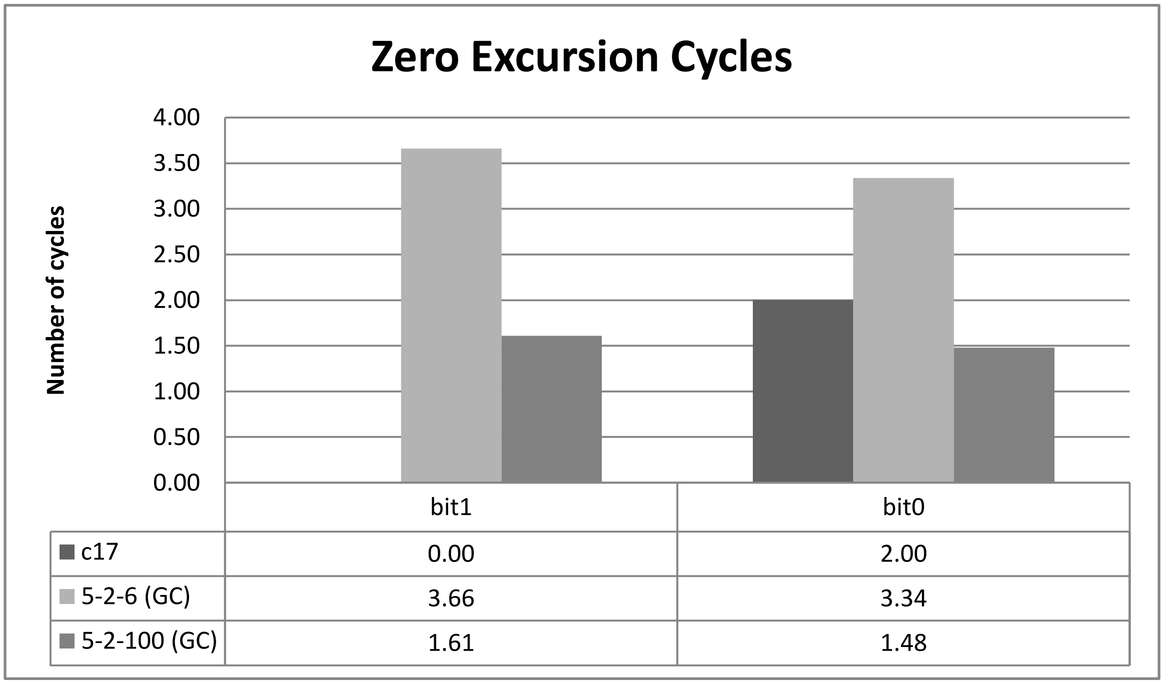
	bit1	bit0
■ c17	0.56	0.56
■ 5-2-6 (GC)	0.50	0.50
■ 5-2-100 (GC)	0.51	0.49



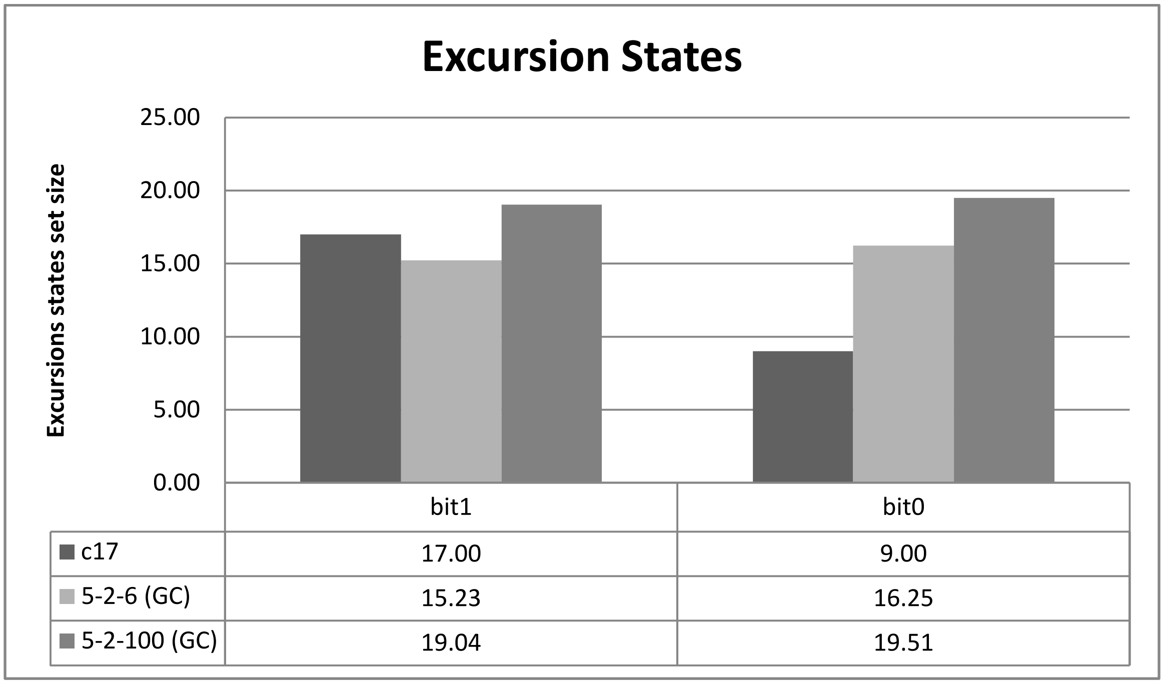
	bit1	bit0
■ c17	16.00	8.00
■ 5-2-6 (GC)	8.00	8.00
■ 5-2-100 (GC)	8.00	8.00



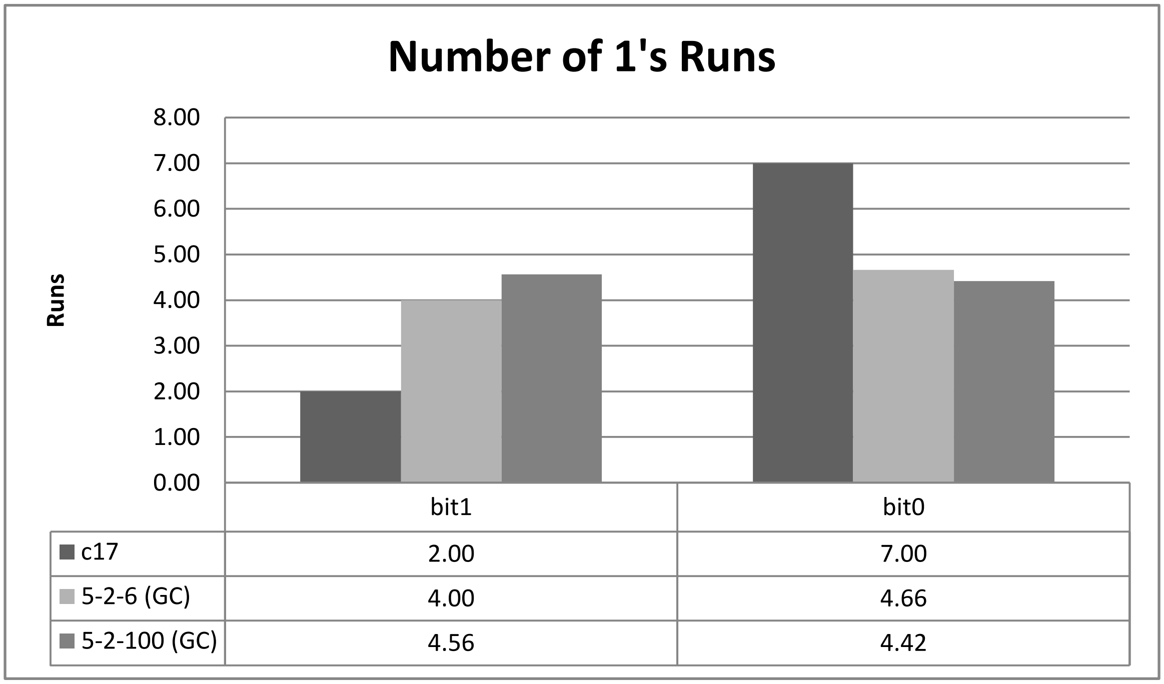
	bit1	bit0
■ c17	8.00	6.00
■ 5-2-6 (GC)	13.98	14.52
■ 5-2-100 (GC)	17.86	18.31



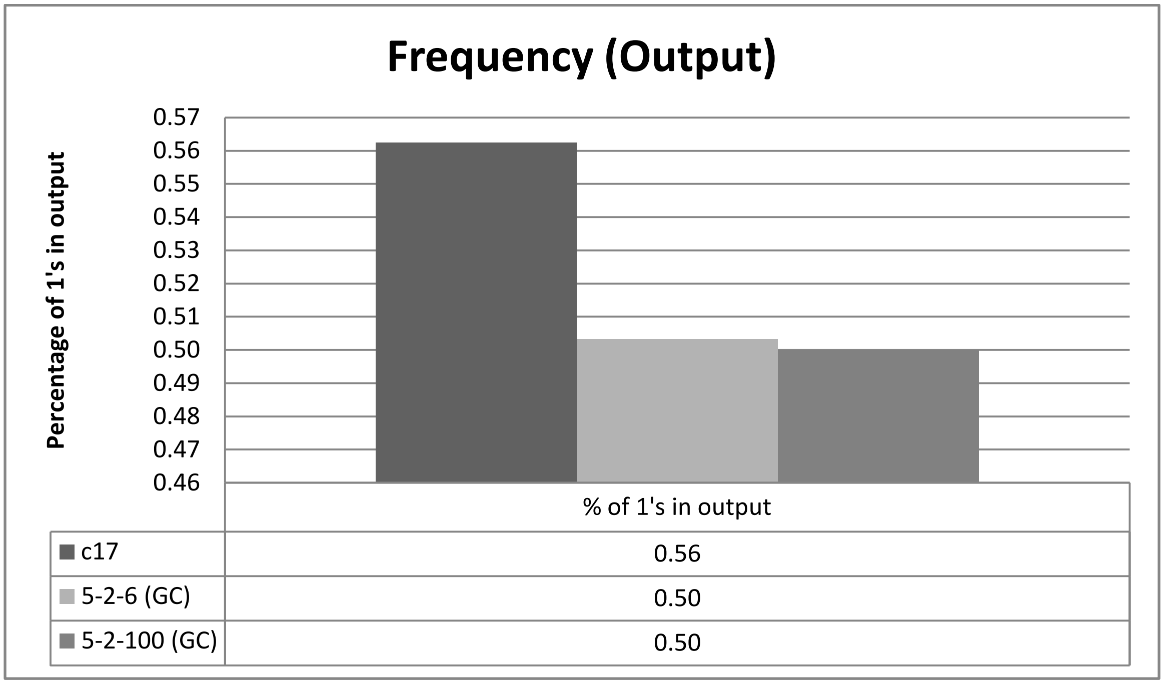
	bit1	bit0
■ c17	0.00	2.00
■ 5-2-6 (GC)	3.66	3.34
■ 5-2-100 (GC)	1.61	1.48



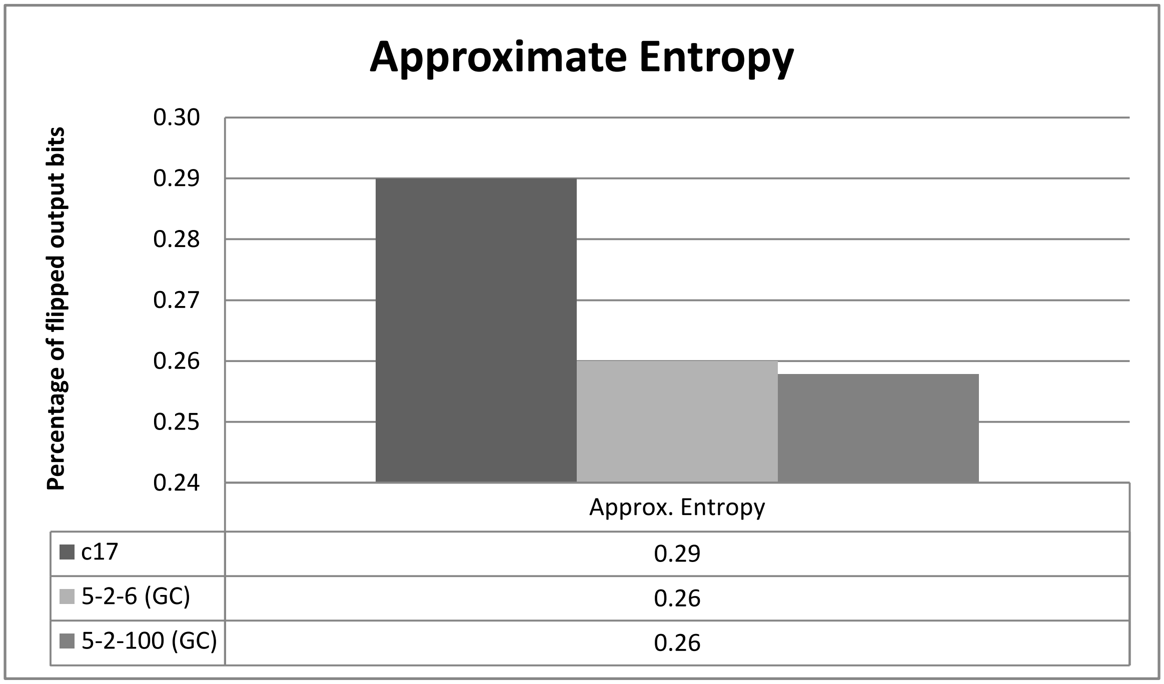
	bit1	bit0
■ c17	17.00	9.00
■ 5-2-6 (GC)	15.23	16.25
■ 5-2-100 (GC)	19.04	19.51



	bit1	bit0
■ c17	2.00	7.00
■ 5-2-6 (GC)	4.00	4.66
■ 5-2-100 (GC)	4.56	4.42

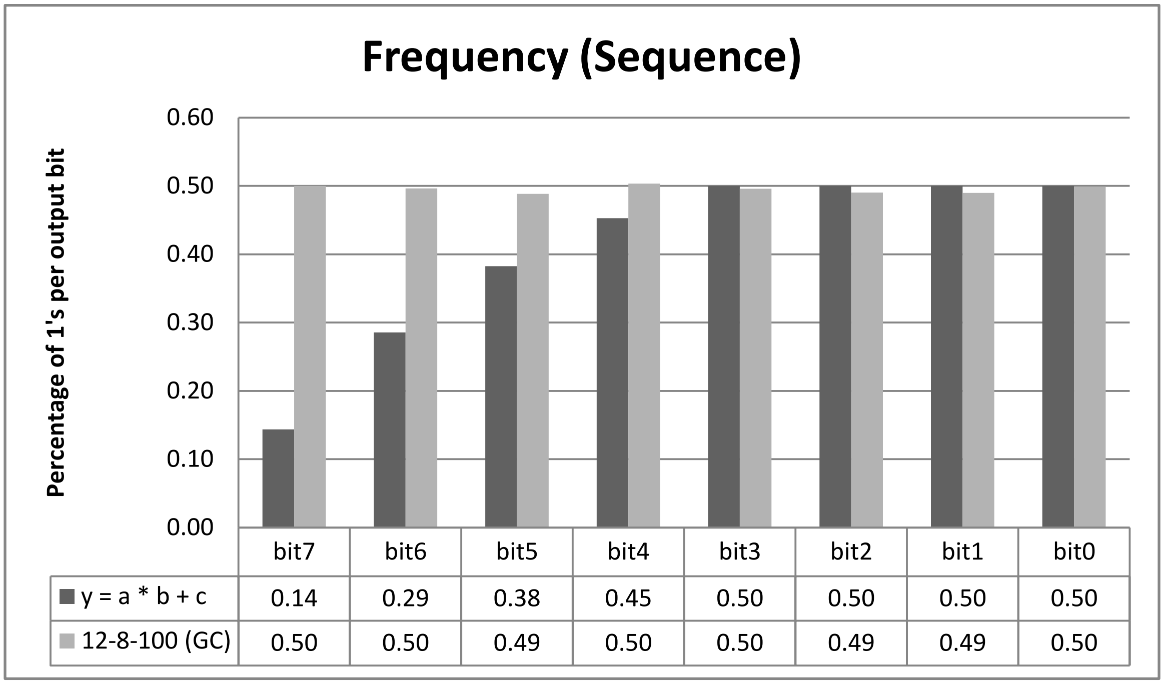


	% of 1's in output
■ c17	0.56
■ 5-2-6 (GC)	0.50
■ 5-2-100 (GC)	0.50

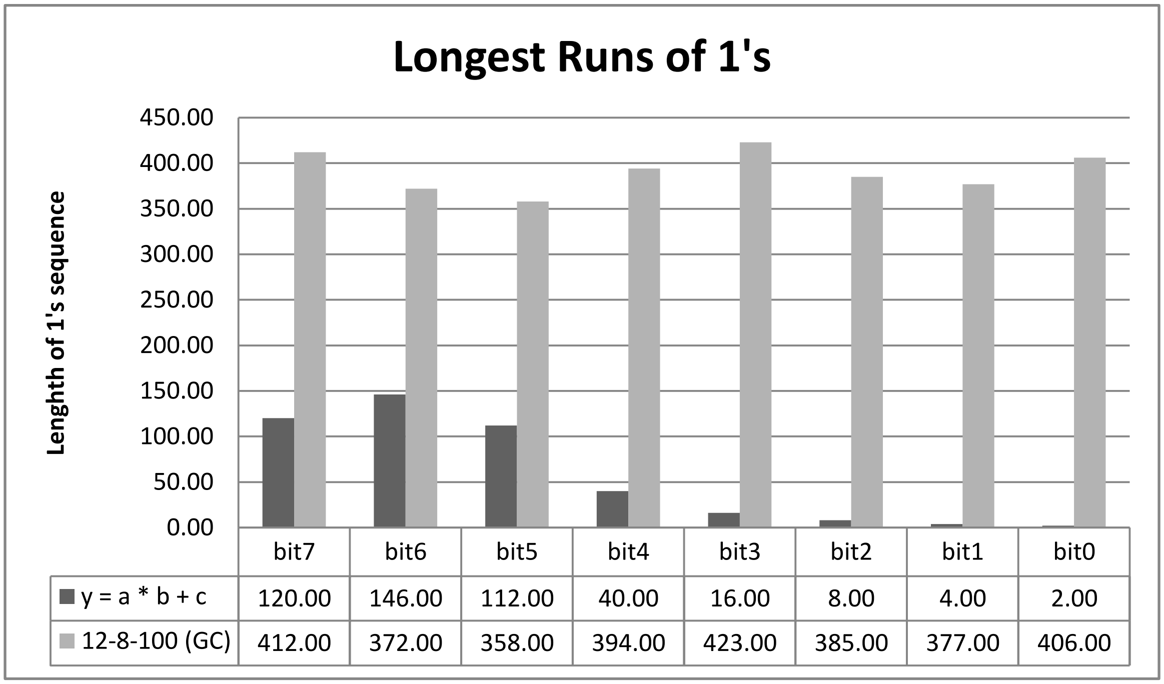


	Approx. Entropy
■ c17	0.29
■ 5-2-6 (GC)	0.26
■ 5-2-100 (GC)	0.26

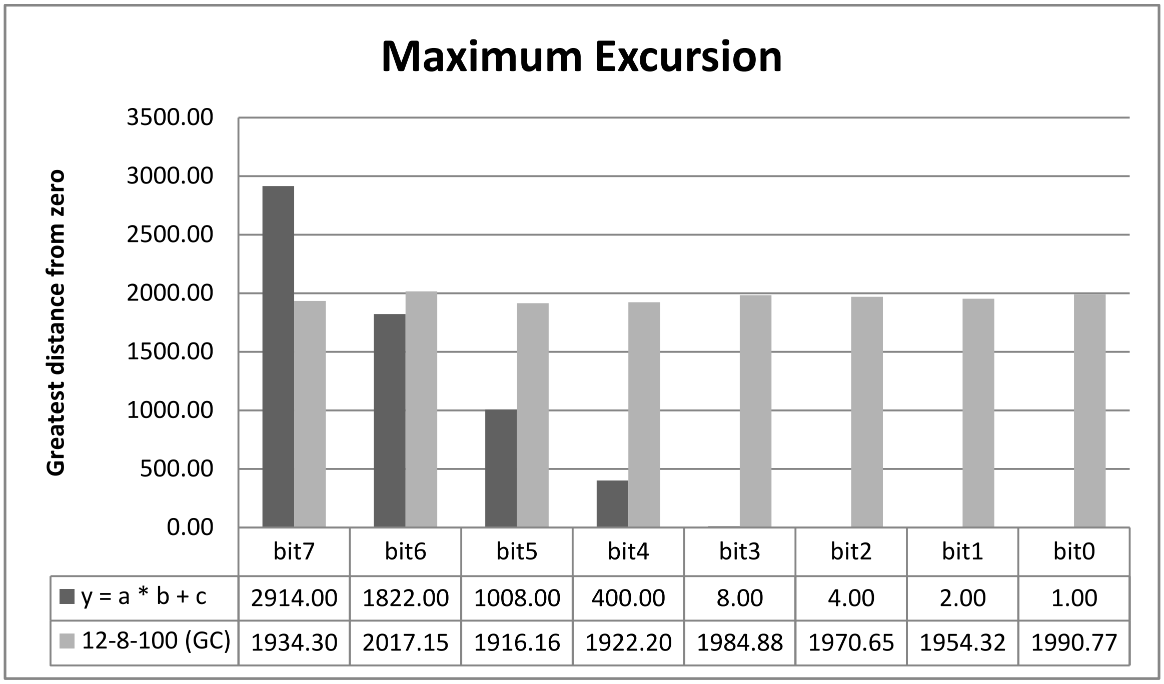
Appendix B: Black-box Analysis of *y = a * b + c* Against Random Functions



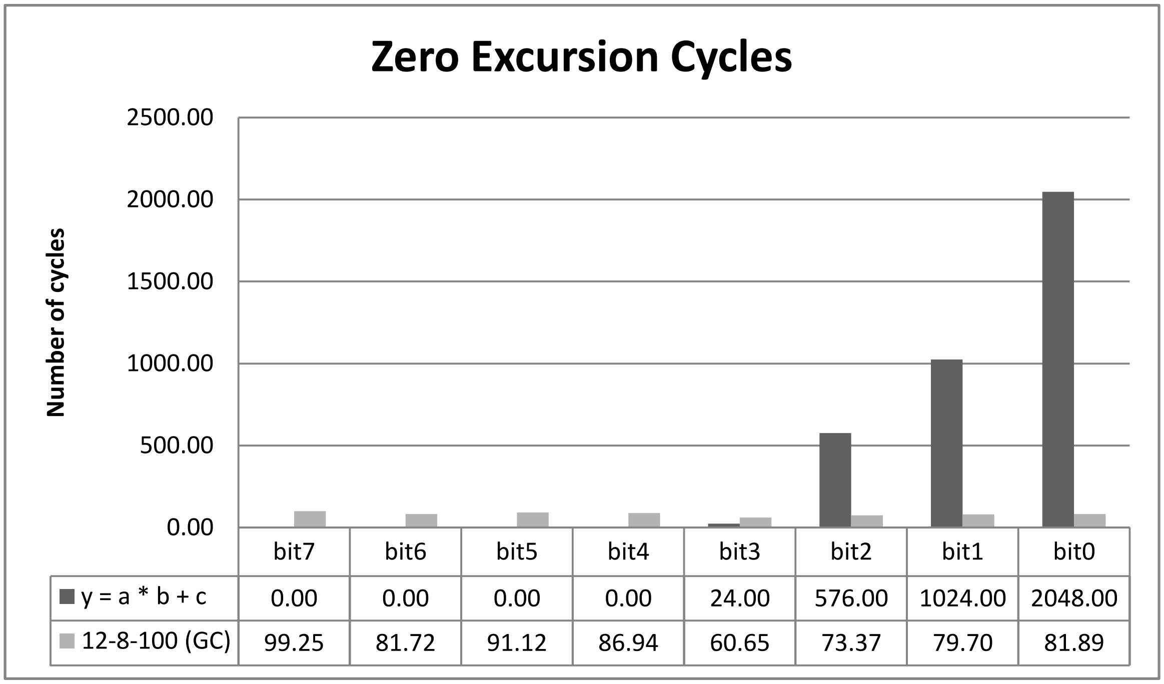
	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
■ y = a * b + c	0.14	0.29	0.38	0.45	0.50	0.50	0.50	0.50
■ 12-8-100 (GC)	0.50	0.50	0.49	0.50	0.50	0.49	0.49	0.50



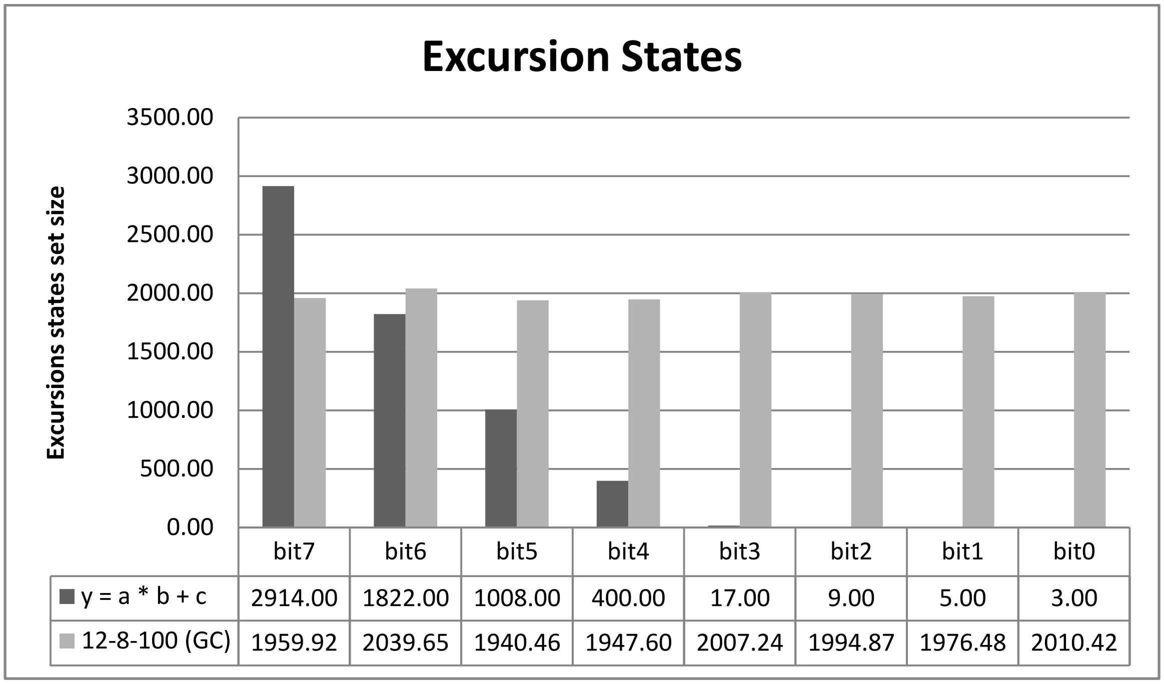
	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
■ y = a * b + c	120.00	146.00	112.00	40.00	16.00	8.00	4.00	2.00
■ 12-8-100 (GC)	412.00	372.00	358.00	394.00	423.00	385.00	377.00	406.00



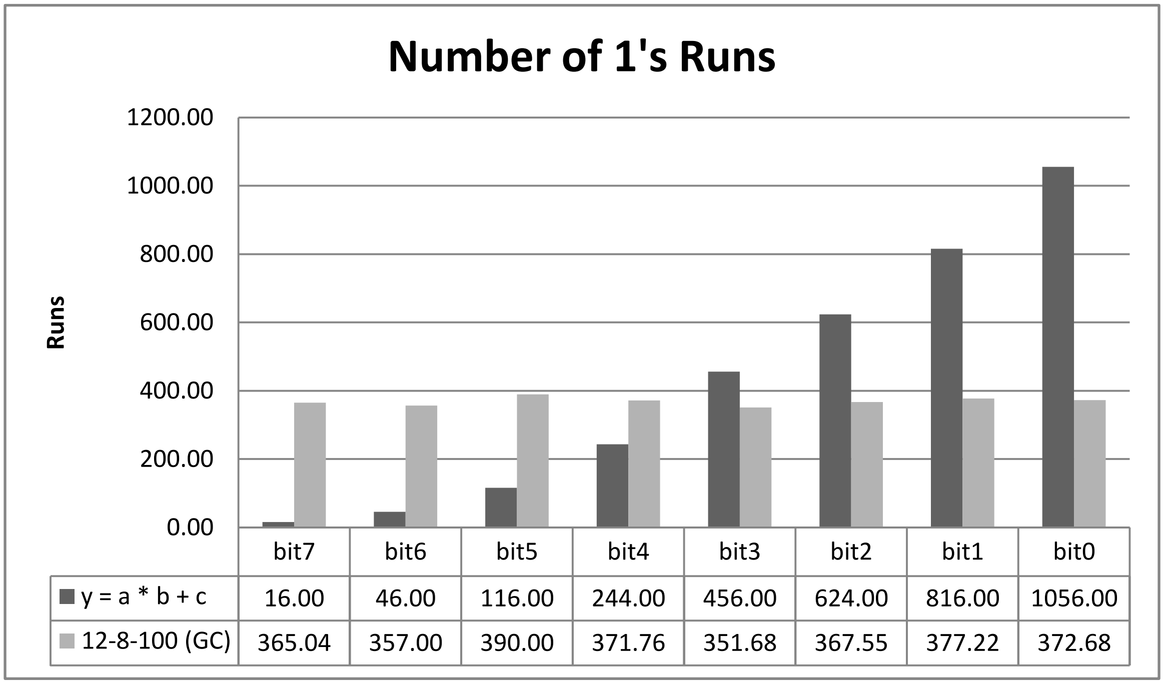
	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
■ y = a * b + c	2914.00	1822.00	1008.00	400.00	8.00	4.00	2.00	1.00
▨ 12-8-100 (GC)	1934.30	2017.15	1916.16	1922.20	1984.88	1970.65	1954.32	1990.77



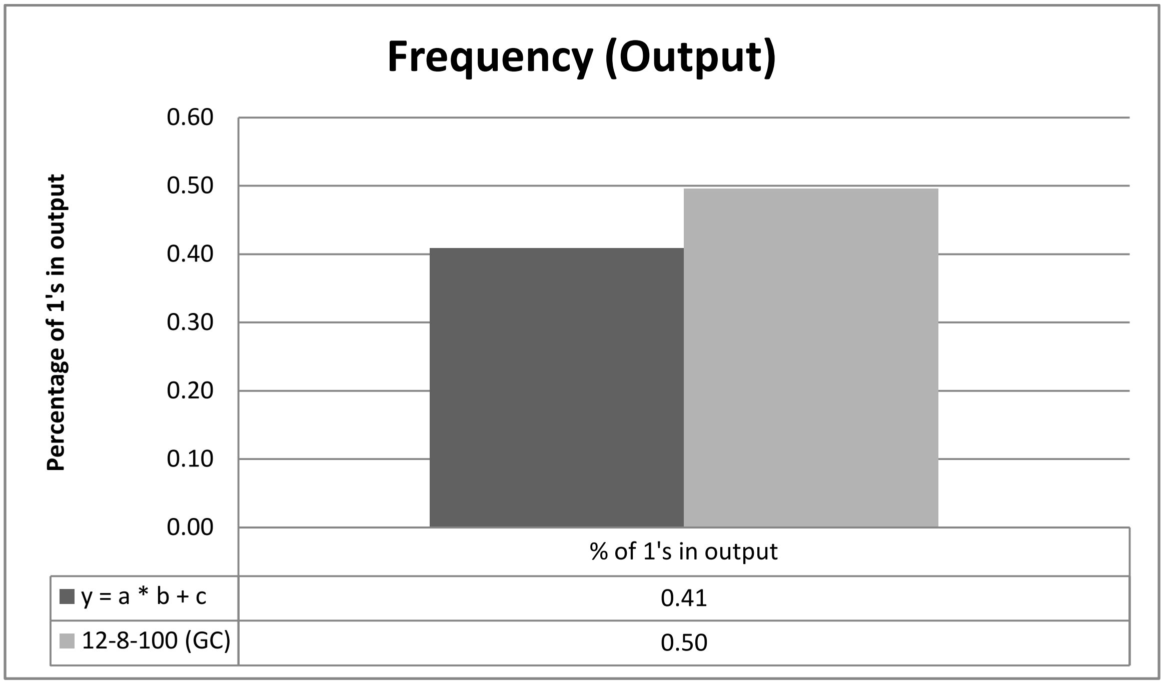
	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
■ y = a * b + c	0.00	0.00	0.00	0.00	24.00	576.00	1024.00	2048.00
▨ 12-8-100 (GC)	99.25	81.72	91.12	86.94	60.65	73.37	79.70	81.89

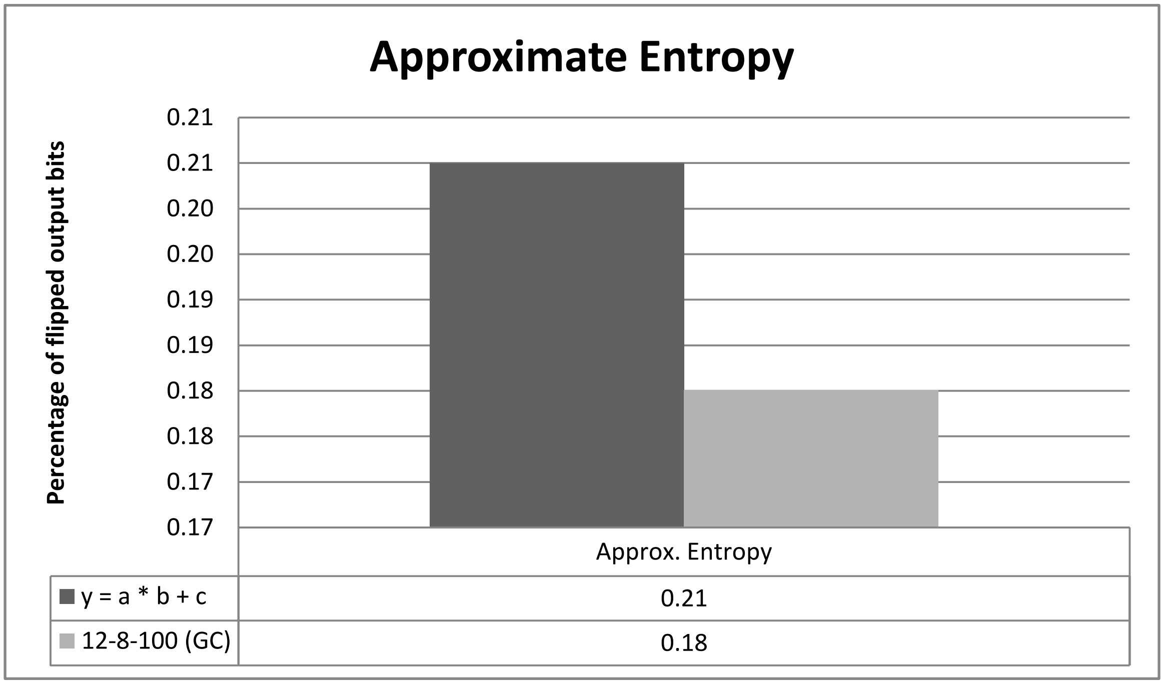


	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
■ y = a * b + c	2914.00	1822.00	1008.00	400.00	17.00	9.00	5.00	3.00
▨ 12-8-100 (GC)	1959.92	2039.65	1940.46	1947.60	2007.24	1994.87	1976.48	2010.42

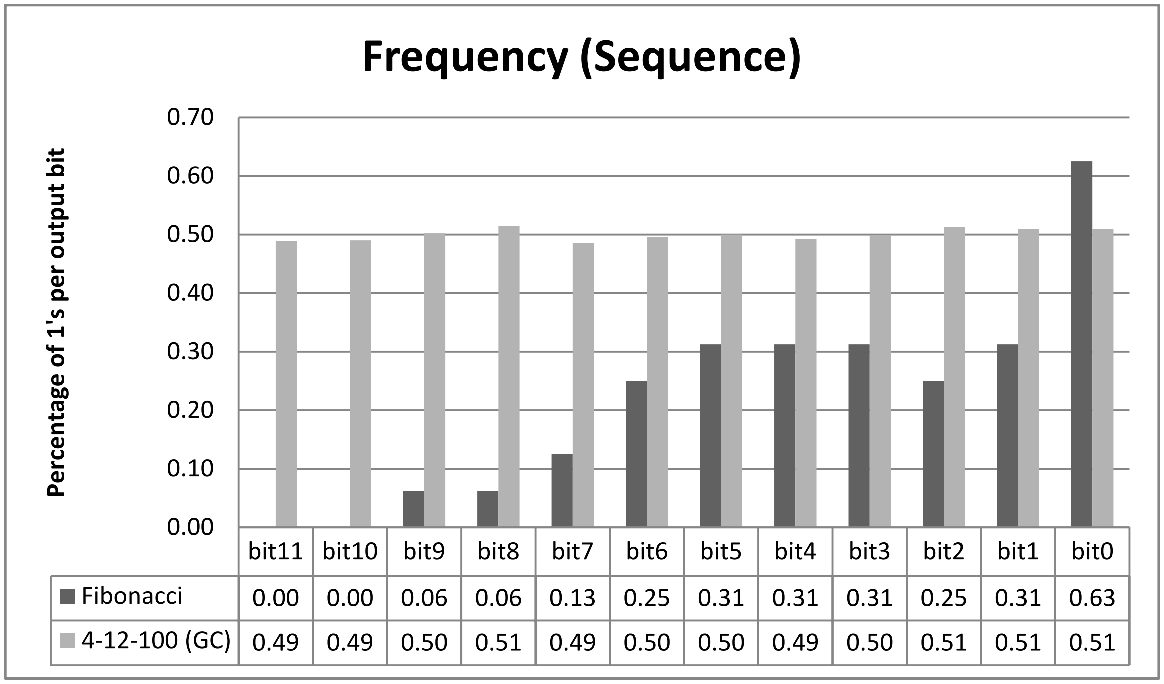


	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
■ y = a * b + c	16.00	46.00	116.00	244.00	456.00	624.00	816.00	1056.00
▨ 12-8-100 (GC)	365.04	357.00	390.00	371.76	351.68	367.55	377.22	372.68

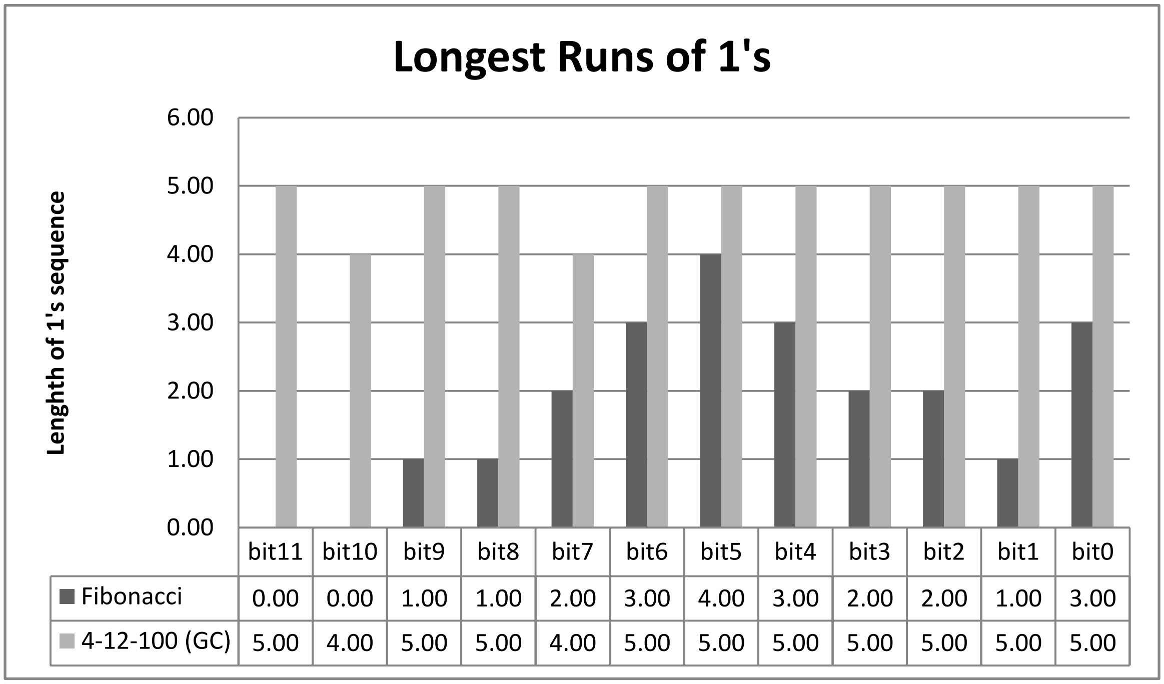

Frequency (Output)
Percentage of 1's in output
0.60
0.50
0.40
0.30
0.20
0.10
0.00
% of 1's in output
■ y = a * b + c
0.41
■ 12-8-100 (GC)
0.50


Approximate Entropy
Percentage of flipped output bits
0.21
0.21
0.20
0.20
0.19
0.19
0.18
0.18
0.17
0.17
Approx. Entropy
■ y = a * b + c
0.21
■ 12-8-100 (GC)
0.18

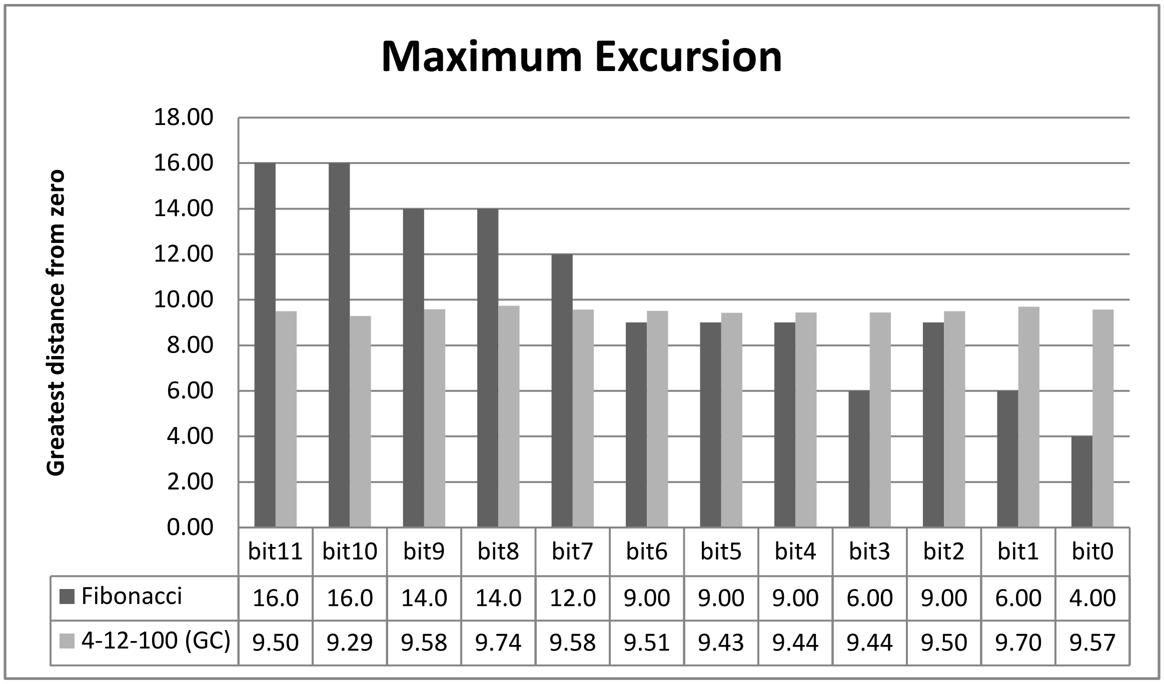
Appendix C: Black-box Analysis of Fibonacci Against Random Functions



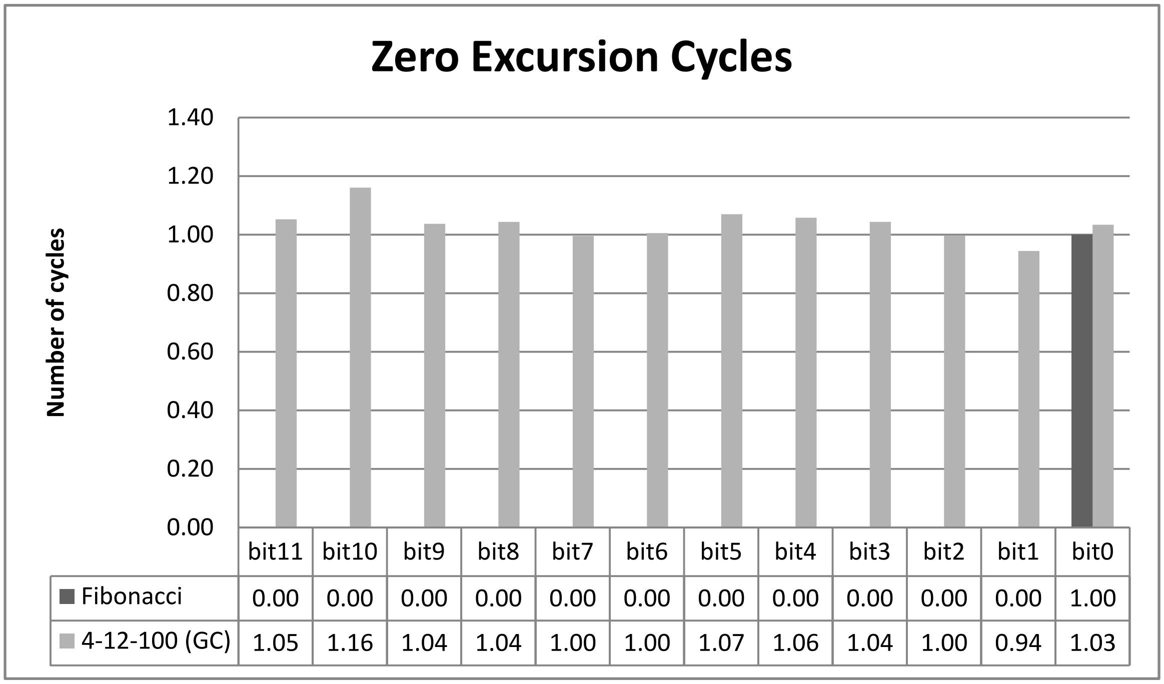
	bit11	bit10	bit9	bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
■ Fibonacci	0.00	0.00	0.06	0.06	0.13	0.25	0.31	0.31	0.31	0.25	0.31	0.63
■ 4-12-100 (GC)	0.49	0.49	0.50	0.51	0.49	0.50	0.50	0.49	0.50	0.51	0.51	0.51



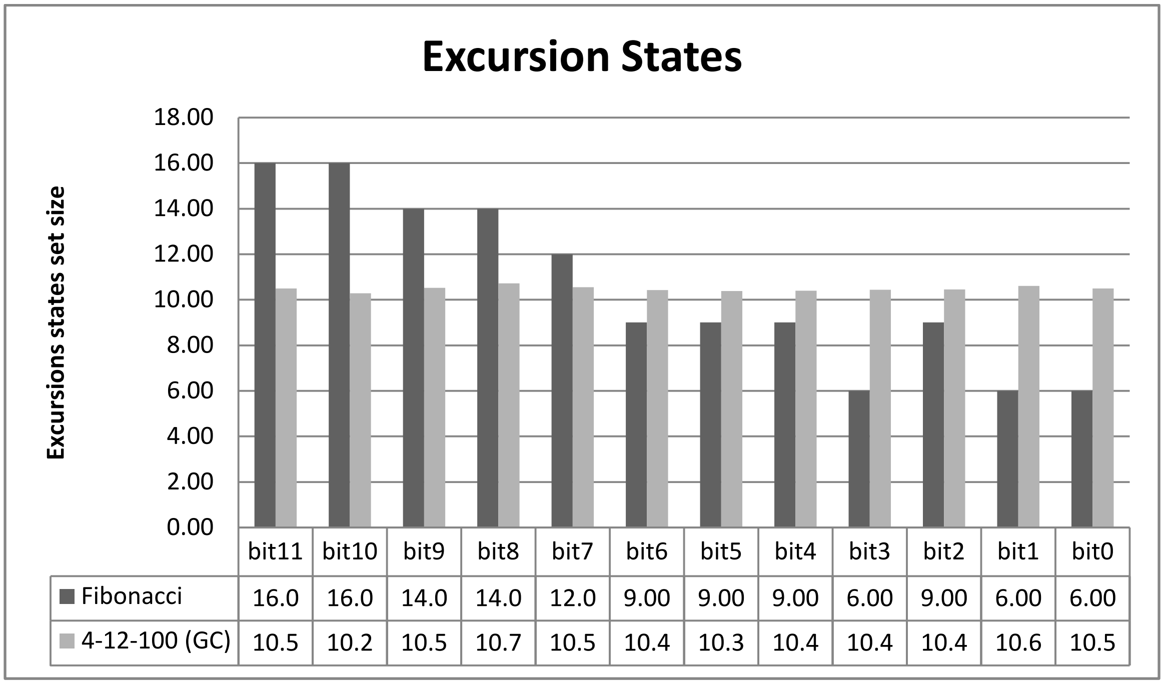
	bit11	bit10	bit9	bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
■ Fibonacci	0.00	0.00	1.00	1.00	2.00	3.00	4.00	3.00	2.00	2.00	1.00	3.00
■ 4-12-100 (GC)	5.00	4.00	5.00	5.00	4.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00



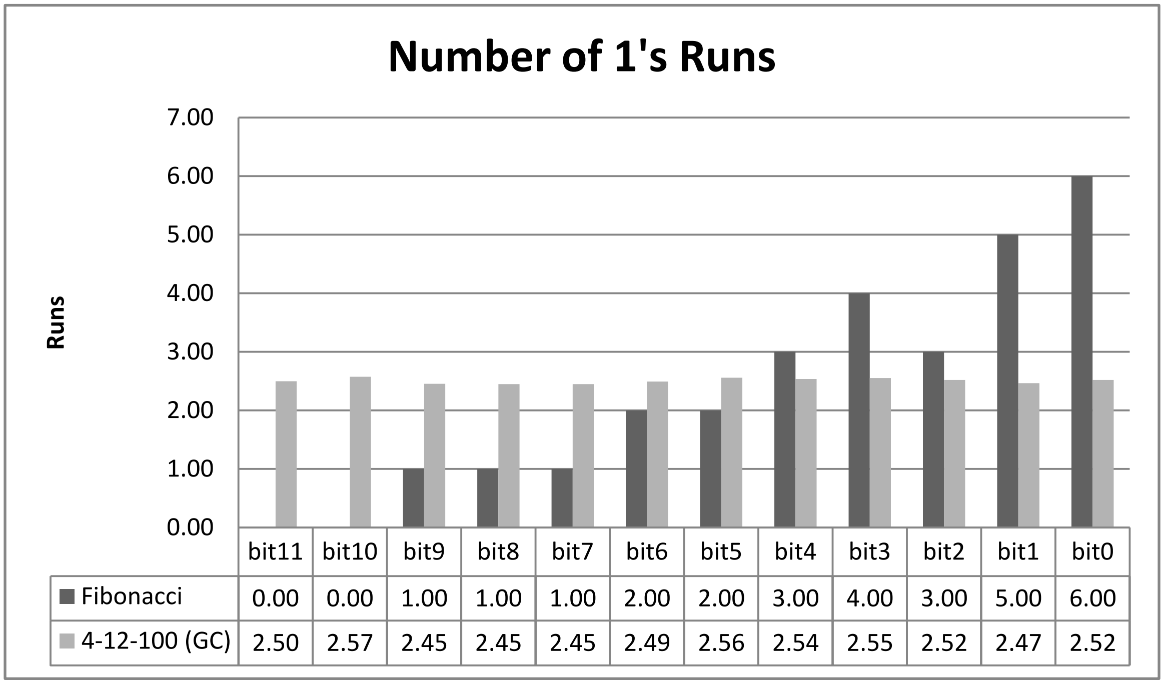
	bit11	bit10	bit9	bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
■ Fibonacci	16.0	16.0	14.0	14.0	12.0	9.00	9.00	9.00	6.00	9.00	6.00	4.00
□ 4-12-100 (GC)	9.50	9.29	9.58	9.74	9.58	9.51	9.43	9.44	9.44	9.50	9.70	9.57



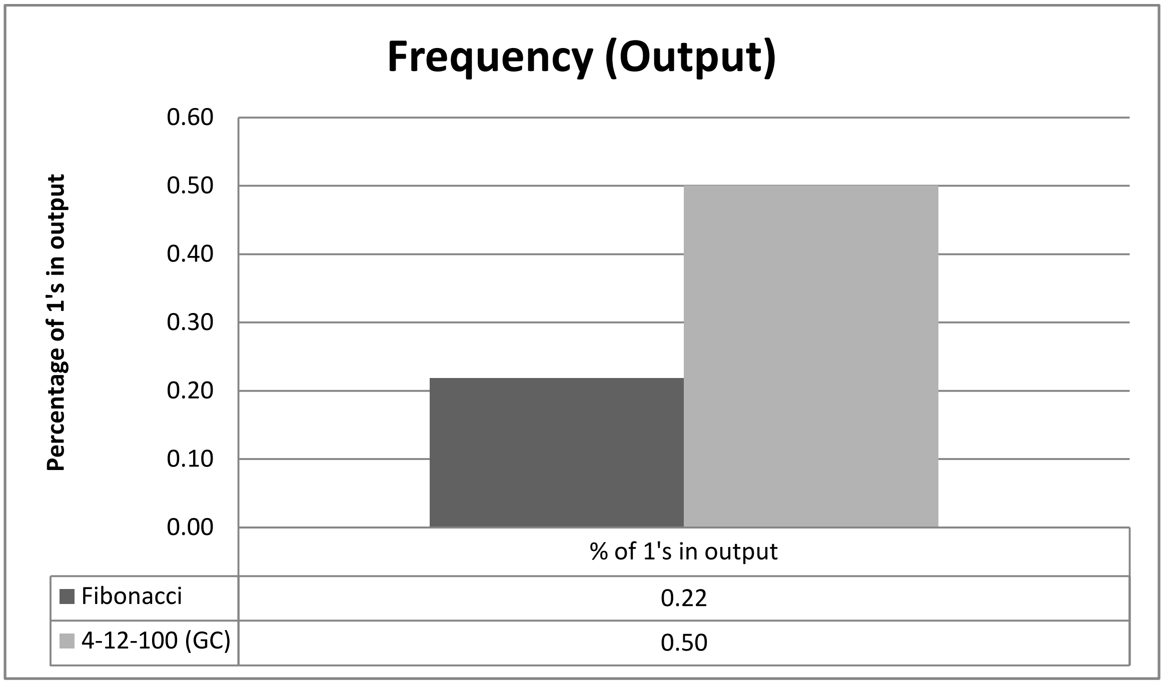
	bit11	bit10	bit9	bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
■ Fibonacci	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
□ 4-12-100 (GC)	1.05	1.16	1.04	1.04	1.00	1.00	1.07	1.06	1.04	1.00	0.94	1.03

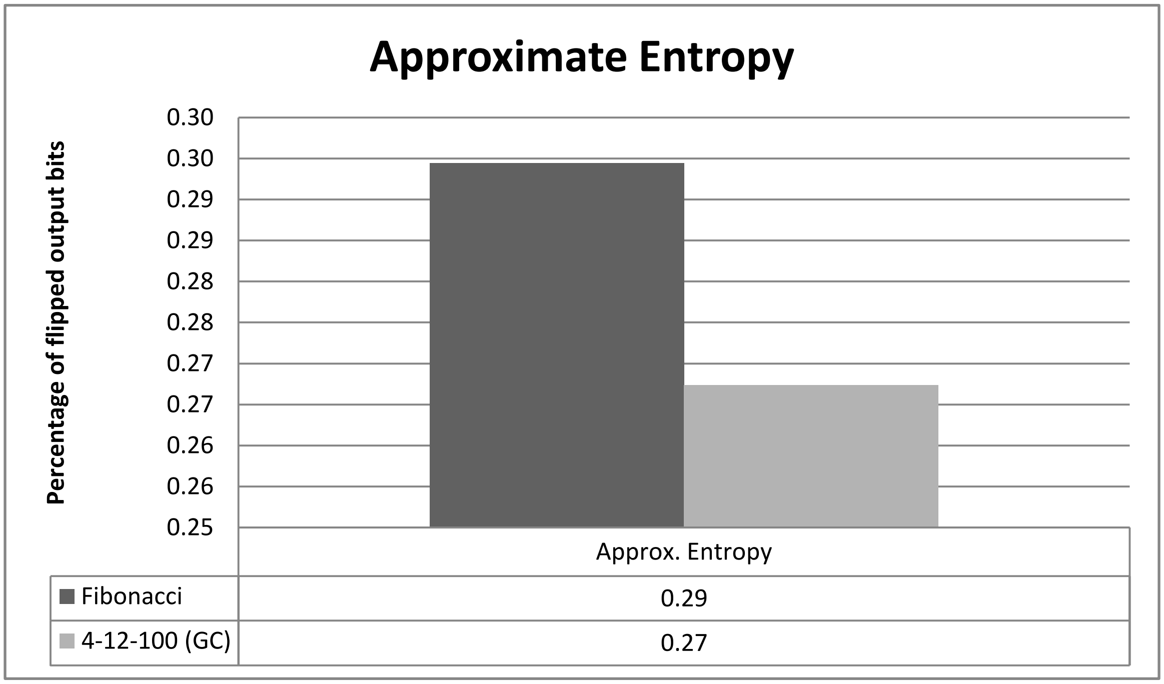


	bit11	bit10	bit9	bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
■ Fibonacci	16.0	16.0	14.0	14.0	12.0	9.00	9.00	9.00	6.00	9.00	6.00	6.00
■ 4-12-100 (GC)	10.5	10.2	10.5	10.7	10.5	10.4	10.3	10.4	10.4	10.4	10.6	10.5



	bit11	bit10	bit9	bit8	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
■ Fibonacci	0.00	0.00	1.00	1.00	1.00	2.00	2.00	3.00	4.00	3.00	5.00	6.00
■ 4-12-100 (GC)	2.50	2.57	2.45	2.45	2.45	2.49	2.56	2.54	2.55	2.52	2.47	2.52


Frequency (Output)
Percentage of 1's in output
0.60
0.50
0.40
0.30
0.20
0.10
0.00
% of 1's in output
■ Fibonacci
0.22
■ 4-12-100 (GC)
0.50


Approximate Entropy
Percentage of flipped output bits
0.30
0.30
0.29
0.29
0.28
0.28
0.27
0.27
0.26
0.26
0.25
Approx. Entropy
■ Fibonacci
0.29
■ 4-12-100 (GC)
0.27

Bibliography

"Alice and Bob," Excerpt from unpublished article. n. pag. http://en.wikipedia.org/wiki/Alice_and_Bob 12 December 2007.

Algesheimer, Joy, Christian Cachin, Jan Camenisch, and Gunter Karjoth, G. "Cryptographic Security for Mobile Code," *Security and Privacy,* 2001, IEEE 2001:2-11 (14 May 2001).

Barak, Boaz, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. "On the (Im)possibility of Obfuscating Programs," *Electronic Colloquium on Computational Complexity,* 57: 1-42 (August 2001).

Bassham, Lawrence E. "The Advanced Encryption Standard Algorithm Validation Suite (AESAVS)," http://csrc.nist.gov/groups/STM/cavp/documents/aes/AESAVS.pdf 15 November 2002.

"Block Cipher Modes of Operation," Excerpt from unpublished article. n. pag. http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation 12 Aug 2007.

Bellare, Mihir and Phillip Rogaway, "Random Oracles are Practical: A Paradigm for Designing Efficient Protocols." http://coblitz.codeen.org:3125/citeseer.ist.psu.edu/cache/papers/cs/2104/http:zSzzSzwwwcsif.cs.ucdavis.eduzSz~rogawayzSzpaperszSzoracle.pdf/bellare95random.pdf 14 February 1995.

"B-29 Superfortress," Excerpt from unpublished article. n. pag. http://commons.wikimedia.org/wiki/B-29_Superfortress 12 August 2007.

Canetti, Ran, Oded Goldreich, and Shai Halevi. "The Random Oracle Methodology, Revisited," http://arxiv.org/PS_cache/cs/pdf/0010/0010019v1.pdf 17 May 2006.

Cappaert, Jan, Brecht Wyseur, and Bart Preneel. "Software Security Techniques." COSIC Internal Report. http://www.cosic.esat.kuleuven.be/publications/article-554.pdf October 2004.

Carlson, Bruce. Director of Operational Requirements, USAF. "Stealth Fighters," News Transcript. 20 April 1999. http://www.defenselink.mil/utility/printitem.aspx?print=http://www.defenselink.mil/transcripts/transcript.aspx?transcriptid=597 16 Aug 2007.

Chow, Stanley, Phil Eisen, Harold Johnson and Paul van Oorschot. "White-Box Cryptography and an AES Implementation," *Record of the 9th Annual Workshop on Selected Areas in Cryptography,* LNCS 2595: 250-270 (March 2002).

Christiansen, Bradley D., Yong C. Kim, Robert W. Bennington and Christopher J. Ristich. "Decoy Circuits for FPGA Design Protection," *Field Programmable Technology 2006,* 10.1109: 373-376 (December 2006).

Collberg, Christian S., Clark Thomborson and Douglas Low. "A Taxonomy of Obfuscating Transforms," *Technical Report 148, Department of Computer Science, University of Auckland,* 148: 1-36 (July 1997).

Collberg, Christian S. and Clark Thomborson. "Watermarking, Tamper-Proofing, and Obfuscation—Tools for Software Protection," *IEEE Transactions on Software Engineering,* 28.8: 735-746 (August 2002).

"Content Scramble System," Excerpt from unpublished article. n. pag. http://en.wikipedia.org/wiki/Content_Scramble_System 12 August 2007.

"Data Encryption Standard," Excerpt from unpublished article. n. pag. http://en.wikipedia.org/wiki/Data_Encryption_Standard 12 August 2007.

Department of Justice. *Former Chinese National Charged with Stealing Military Application Trade Secrets From Silicon Valley Firm to Benefit Governments of Thailand, Malaysia, and China.* 14 December 2006 http://www.usdoj.gov/usao/can/2006/2006_12_14_meng.indictment.press.html 16 August 2007.

"Digital Rights Management," Excerpt from unpublished article. n. pag. http://en.wikipedia.org/wiki/Digital_rights_management 12 August 2007.

Eilam, Eldad. *Reversing: Secrets of Reverse Engineering.* Indianapolis: Wiley Publishing, 2005.

Goldwasser, Shafi and Guy N. Rothblum. "On Best-Possible Obfuscation," *Lecture Notes in Computer Science*, 4392: 194-213 (May 2007).

Hofheinz, Dennis, John Malone-Lee, and Martijin Stam. "Obfuscation for Cryptographic Purposes," *Lecture Notes in Computer Science*, 4392: 214-232 (May 2007).

Hohenberger, Susan and Guy N. Rothblum. "Securely Obfuscating Re-Encryption," http://www.cs.jhu.edu/~susan/papers/HRSV07.pdf (14 February 2007).

Hughes, Jeff, and Martin R. Stytz. "Advancing Software Security—The Software Protection Initiative," http://www.preemptive.com/documentation/SPI_software_Protection_Initative.pdf 1 September 2007.

Jorstad, Norman D. "Cryptographic Algorithm Metrics," http://csrc.nist.gov/nissc/1997/proceedings/128.pdf (07 October 1997).

Loureiro, Sergio, Laurent Bussard, and Yves Roudier. "Extending Tamper-Proof Hardware Security to Untrusted Execution Environments," *Proceedings of the 5th conference on Smart Card Research and Advanced Application Conference*. 5:1-12 http://www.eurecom.fr/~nsteam/Papers/cardis02.pdf November 2002.

"Kerchkhoff's Principle," Excerpt from unpublished article. n. pag. http://en.wikipedia.org/wiki/Kerckhoffs'_principle 10 July 2007.

National Institute of Standards and Technology. *A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications.* NIST Special Publication 800-22. Gaithersburg MD, 15 May 2001 http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22b.pdf.

McDonald, J. Todd and Alec Yasinsac. "Applications for Provably Secure Intent Protection with Bounded Input-Size Programs," *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on* , pp.286-293, 10-13 April 2007 http://ieeexplore.ieee.org/iel5/4159773/4159774/04159815.pdf?tp=&isnumber=4159774&arnumber=4159815.

"Plane Called an 'Electronic Vacuum Cleaner' on Routine Mission," *CNN,* 3 April 2001 http://www.archives.cnn.com/2001/US/04/03/plane.mission/index.html 16 August 2007.

Preneel, B., A. Biryukov, E. Oswald, B. Van Rompay, L. Granboulan, E. Dottax, S. Murphy, A. Dent, J. White, M. Dichtl, S. Pyka, M. Schafheutle, P. Serf, E. Biham, E. Barkan, O. Dunkelman, J.-J. Quisquarter, M. Ciet, F. Sica, L. Knudsen, M. Parker, and H. Raddum. "NESSIE Security Report Version 2.0," https://www.cosic.esat.kuleuven.be/nessie/deliverables/D20-v2.pdf (19 February 2003).

"Quine-McCluskey Algorithm," Excerpt from unpublished article. n. pag. http://en.wikipedia.org/wiki/Quine-McCluskey_algorithm 12 Aug 2007.

"Reverse Engineering," Excerpt from unpublished article. n. pag. http://en.wikipedia.org/wiki/Reverse_engineering 10 July 2007.

Richelson, Jeffery T. "When Secrets Crash," *Air Force Magazine,* 7.01:58+ (July 2001). http://www.afa.org/magazine/july2001/0701crash.pdf 16 August 2007.

Sander, Thomas and Christian F. Tschudin. "Protecting Mobile Agents Against Malicious Hosts," http://citeseer.ist.psu.edu/cache/papers/cs/16015/http:zSzzSzwww.icsi.berkeley.eduzSz~tschudinzSzpszSzma-security.pdf/sander98protecting.pdf February 1998.

Sander, Thomas and Christian F. Tschudin. "On Software Protection Via Function Hiding," http://citeseer.ist.psu.edu/cache/papers/cs/14081/http:zSzzSzwww.informatik.uni-stuttgart.dezSzipvrzSzvszSzpersonenzSz..zSzlehrezSzws9899zSzvorlesungenzSzMAzSzSanTsc98.pdf/sander98software.pdf December 1998.

"Software Agent," Excerpt from unpublished article. n. pag. http://en.wikipedia.org/wiki/Software_agent 12 Aug 2007.

Thomborson, Clark. A Class handout. "Methods for Software Protection," http://www.cs.auckland.ac.nz/~cthombor/Pubs/TC/SWprot7Apr07.ppt 07 April 07.

Torri, Stephen, John A. Hamilton Jr., Derek Sanders, and Gordon Evans. "A Primer on Java Obfuscation," http://www.stsc.hill.af.mil/crosstalk/2007/12/0712TorriSandersHamiltonEvans.html December 2007.

Travis, Greg. "How to Lock Down Your Java Code (Or Open Up Someone Else's)," http://www.ibm.com/developerworks/java/library/j-obfus/?loc=tstheme 01 May 2001.

"Tupolev Tu-4," Excerpt from unpublished article. n. pag. http://en.wikipedia.org/wiki/Tupolev_Tu-4 12 Aug 2007.

"U.S. Arms Software Export Guilty Plea Marks First," Reuters, 2 August 2007 http://www.reuters.com/articlePrint?articleId=USN0246288320070802 16 August 2007.

Vahid, Frank. "It's Time to Stop Calling Circuits 'Hardware'," *Computer* 40.9:106-108. http://ieeexplore.ieee.org/iel5/2/4302594/04302628.pdf?arnumber=4302628&htry=4 September 2007.

Vita

1Lt Alan C. Lin graduated from John P. Stevens High School in Edison, New Jersey. He entered undergraduate studies at Rutgers University's School of Engineering in Piscataway, New Jersey where he graduated with a Bachelors of Science degree in Computer Engineering in January 2004. He was commissioned through Officer Training School at Maxwell AFB in April 2004.

His first assignment was at Hanscom AFB as a developmental engineer working on basic research for biometric security at Air Force Research Laboratory, Sensors Directorate, Electro-Optics Branch, in April 2004. In August 2006, he entered the Graduate School of Engineering and Management, Air Force Institute of Technology. Upon graduation, he will be assigned to Los Angeles AFB, Space and Missile Systems Center, Concept Engineering Branch.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 074-0188*

1. REPORT DATE *(DD-MM-YYYY)* 27-03-2008	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From – To)* June 2007 – March 2008

4. TITLE AND SUBTITLE Software Obfuscation with Symmetric Cryptography	**5a. CONTRACT NUMBER** **5b. GRANT NUMBER** **5c. PROGRAM ELEMENT NUMBER**
6. AUTHOR(S) Lin, Alan C., 1Lt, USAF	**5d. PROJECT NUMBER** 08-183 **5e. TASK NUMBER** **5f. WORK UNIT NUMBER**
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way, Building 640 WPAFB OH 45433-8865	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GCS/ENG/08-15
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Dr. Robert L. Herklotz Program Manager: Security and Information Operations Air Force Office of Scientific Research Suite 325, Room 3112 875 N. Randolph Street Arlington, VA 22203-1768 email- robert.herklotz@afosr.af.mil (703) 696-6565 fax (703) 696-8450	**10. SPONSOR/MONITOR'S ACRONYM(S)** **11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Software protection is of great interest to commercial industry. Millions of dollars and years of research are invested in the development of proprietary algorithms used in software programs. A reverse engineer that successfully reverses another company's proprietary algorithms can develop a competing product to market in less time and with less money. The threat is even greater in military applications where adversarial reversers can use reverse engineering on unprotected military software to compromise capabilities on the field or develop their own capabilities with significantly less resources. Thus, it is vital to protect software, especially the software's sensitive internal algorithms, from adversarial analysis.

Software protection through obfuscation is a relatively new research initiative. The mathematical and security community have yet to agree upon a model to describe the problem let alone the metrics used to evaluate the practical solutions proposed by computer scientists. We propose evaluating solutions to obfuscation under the intent protection model, a combination of white-box and black-box protection to reflect how reverse engineers analyze programs using a combination white-box and black-box attacks. In addition, we explore use of experimental methods and metrics in analogous and more mature fields of study such as hardware circuits and cryptography. Finally, we implement a solution under the intent protection model that demonstrates application of the methods and evaluation using the metrics adapted from the aforementioned fields of study to reflect the unique challenges in a software-only software protection technique.

15. SUBJECT TERMS

Software metrics, cryptography, information assurance, software tools

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18.NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** J. Todd McDonald, Lt Col, USAF
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	108	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-6565, ext 4639 (Jeffrey.McDonald@afit.edu)